L Number	Hits	Search Text	DB	Time stamp
-	5069	(process$3 with access$3) same layer$1	USPAT; US-PGPUB	2003/11/14 15:46
-	1255	(process$3 near5 layer$1) same (access$3 near5 layer$1)	USPAT; US-PGPUB	2003/11/14 15:47
-	778	((process$3 near5 layer$1) same (access$3 near5 layer$1)) and object$1	USPAT; US-PGPUB	2003/11/14 15:49
-	441	(((process$3 near5 layer$1) same (access$3 near5 layer$1)) and object$1) and @ad<20000425	USPAT; US-PGPUB	2003/11/14 15:54
-	441	(((process$3 near5 layer$1) same (access$3 near5 layer$1)) and object$1) and @ad<20000425	USPAT; US-PGPUB	2003/11/14 15:55
-	115	((((process$3 near5 layer$1) same (access$3 near5 layer$1)) and object$1) and @ad<20000425) and class$2	USPAT; US-PGPUB	2003/11/14 15:55

US-PAT-NO: **6269396**

DOCUMENT-IDENTIFIER: US 6269396 B1

TITLE: Method and platform for interfacing between application programs performing telecommunications functions and an operating system

---------- KWIC ---------

US Patent No. - PN (1):
6269396

Application Filing Date - AD (1):
19981211

Brief Summary Text - BSTX (4):
In one aspect of the present invention, a telecom platform forming an interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network. The telecom platform includes network management processes operable to provide inter-node configuration, monitoring and management functionality, node management processes operable to provide node initialization, configuration, monitoring, and management functionality, event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events, common processes operable to provide a library of a plurality of programming tools for the development of the application programs, communications processes operable to provide message handling functionality, and distributed **object** processes operable to provide a distributed database repository for **object**-based communications.

Brief Summary Text - BSTX (5):
In another aspect of the present invention, a method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network is provided. The method includes supplying network management processes operable to provide inter-node configuration, monitoring and management functionality, supplying node management processes operable to provide node initialization, configuration, monitoring, and management functionality, supplying event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events, supplying common processes operable to provide a library of a plurality of programming tools for the development of the application

programs, supplying communications processes operable to provide message handling functionality, and supplying distributed **object** processes operable to provide a distributed database repository for **object**-based communications.

Drawing Description Text - DRTX (29):
FIG. 21 is a simplified block diagram of the distributed **object** messaging environment according to an embodiment of the present invention;

Drawing Description Text - DRTX (30):
FIG. 22 is a simplified block diagram of the internal debugging and tracing **object** relations according to an embodiment of the present invention;

Detailed Description Text - DETX (4):
As shown in FIG. 1, telecom platform 10 is comprised of three distinct software layers 14-16. Layer #1 is a telecom platform application programming interface (API) layer 14; layer #2 is a telecom platform services layer 15; and layer #3 is a systems interface layer 16. Telecom platform API **layer 14 provides the communication methods for accessing telecom platform services layer** 15, which is comprised of telecommunications middleware services. Telecom platform services layer 15 is the software layer that provides the most commonly needed middleware services for a UNIX-based telecommunications system, for example. System interface layer 16 is comprised of operating system (OS) API and the network links. System interface **layer 16 defines the functions of process** and thread management, memory management, timers, file system, communication, interface to hardware devices, and other system components. Telecom platform 10 allows higher level client applications 12 to be decoupled from the operating system and network. By using telecom platform 10, developers may write applications without having to master the intricacies of the underlying services, such as the operating system and the network, that perform the work on behalf of the application.

Detailed Description Text - DETX (16):
The telecom platform internal architecture is described from both the logical and physical partitioning perspectives. The logical partitioning decomposes the telecom platform into distinct functional areas as shown in FIG. 4. Each functional area contains a cohesive group of **classes,** which together provide one particular system function. The physical partitioning describes the concrete software and hardware decomposition of the system's context. The services provided by telecom platform 10 may be partitioned into two groups: application services 60 and core services 62. Application services may include services that perform information and problem report (IPR)/alarm 64, statistics 65, dictionary 66, graphical user interface (GUI) 67, and host maintenance simulator (HMS). IPR/alarm services 64 provide a standard mechanism to inform the system user of error conditions and other pertinent system information. Statistics services 65 provides the methods to access system-wide measurement data and to generate reports based on the collected data. Dictionary services 66 provide **classes** that are designed to support data storage (persistent, shared or private) and access to the data. Graphical user interface services 67 provide primitive abstractions for building GUI applications, and access to

system utilities and to the system itself, e.g., xterm window and operating system utility programs. Host maintenance simulator services 75 provide a method of interfacing with the telecom platform when there is only one node within the system or when there is not a host to which to connect. It is through the host that control and operation of the platform is made possible.

Detailed Description Text - DETX (17):

Core services 62 may include services that perform network management 68, node management 69, distributed **object** 70, communications 72, common functions 73, and event handling 74. Network management services 68 directs network activities, e.g., configuration of nodes and network-level fault processing. Node management services 69 directs node-level processes, e.g., node status reporting and link management. Distributed **object** services 70 provide a distributed database repository for **object**-based communication in a multi-processing environment. Communications services 72 provide the mechanism for handling messages across interprocessing links external to the platform. Common services 73 provide a library of programming tools to aid in the rapid development of processes designed to run on or within the telecom platform. Event services 74 provide the capability to initiate, terminate, and/or distribute specific actions significant to a task.

Detailed Description Text - DETX (19):

FIG. 5 further shows the telecom platform services and their dependencies. The developer accesses all of the core and application services through telecom platform application program interfaces 14. The developer may also access the operation system, network, and third party software/hardware if the need arises. Interprocess **object**-based communication is handled by communication services 72. Most of the core and application services dependent on communication services 72 and common services 73 to perform their respective functions. Graphical user interface services 67 may only be dependent on communication services 72. The arrows in FIG. 5 indicate the dependency relationships between the services.

Detailed Description Text - DETX (25):

The **class** name for the network platform manager is NetPM. NetPM is responsible for providing management functionality of the platform resources. The platform is a distributed system consisting of multiple nodes or servers which provide processing power for specific services, such as calling card or credit card validation. The service provided by a server is determined by the configurable elements residing on the node. NetPM manages all the configuration data associated with the platform. Configuration data includes information about the hardware, such as the TCP/IP address of a server, status information, such as server and query status, software configuration information, such as application type, node name, and information relating to the individual configurable elements.

Detailed Description Text - DETX (32):

NetPM uses a NetMAP **object** to manage all the configuration data. NetPM also uses a persistent dictionary to retain server status, query status, and

scheduled actions information across platform manager resets. A Disk File Dictionary **object** is used to manager this dictionary. NetPM is responsible for maintaining the integrity of the configuration data between the two platform manager servers. NetPM uses a persistent dictionary, database equalization, and auditing to maintain the integrity of the data.

Detailed Description Text - DETX (46):

NetPM provides, partially through the use of two alias **objects,** two sets of routing options to other processes wishing to communicate with NetPM. NetPM provides a local, and a global active-standby option. In the local option, all NetPM client requests are sent to the NetPM server **object** in the same node as the client **object**. In the global active-standby option, all NetPM client requests are sent to the globally (i.e. possibly inter-nodal) available active NetPM server **object**.

Detailed Description Text - DETX (48):

NetPM also provides a function to initialize the majority of the Server configuration data. This function expects a ServerInfoMsg **object** as input.

Detailed Description Text - DETX (53):

NetPM is also responsible for time synchronization within the server network. Time synchronization consists of three major parts, as shown in FIG. 7B. The first part is for active platform manager 100 to equalize its local time with the time of the host. This includes converting the host's (110) time into a usable form and informing the NodePMs 112 on platform manager nodes 100 and 102 to perform an adjtime( ) function to adjust their clocks in line with host 110. NetPM 104 also informs the host ticker **class** of the new host time when it receives the time message. An xntp process 120 then synchronizes the application nodes' (121) time with the time of the platform manager nodes 100 and 102. Each of the platform manager nodes 100 and 102 are configured as xntp master sources of time. The xntp daemon slaves 122 on application nodes 121 choose one of the master xntp daemons 120 on platform manager nodes 100 and 102 to keep in synch with. Finally, whenever an unsolicited Set Time message is received from host 110, the network's time is the same as the received time.

Detailed Description Text - DETX (55):

NetPM 104 is an **object** with its own thread of control. After building up its NetMAP lists, NetPM 104 goes into an infinite loop waiting for requests. NetPM 104 notifies ConfigMgr 108 whenever there is a change in the service or query status of a server. NetPM 104 also sends these status changes to all the NodePMs 112 in the platform. NetPM 104 notifies the specific NodePM 112 to enable, or disable, query processing. NetPM 104 provides service status synchronization functionality. NetPM 104 builds up the IPU information for the servers in the platform and passes this information to the specific NodePM 112 in the BootNotify member function. NetPM, in all the configuration requests for degradation of service (i.e. GraceDown, ImmedDown, GraceHalt, and ImmedHalt), notifies the specific NodePM 112 of the desired state of the server. NetPM 104 does several things when a server restore is requested. First, NetPM 104 obtains the current status of the server from the specific

NodePM 112. Second, if the returned status is out-of-service/minimum-software, NetPM 104 sends the specific NodePM 112 the relevant NodeSpecInfo. Third, NetPM 104 sends the relevant configurable element descriptor information to the specific NodePM 112. Lastly, NetPM tells the specific NodePM to restore to service.

Detailed Description Text - DETX (61):

The **class** name of Network System Integrity is NetSI. NetSI 106 manages network system integrity for the platform manager. NetSI 106 receives notifications of server downgrades and communication faults from the NodeSI on the faulted node. NetSI 106 determines what action should be taken based on the data given by NodeSI. If the node indicates a downgrade, NetSI will take the appropriate action to downgrade the node from the network level to the desired downgraded state. If the node indicates a communication fault, NetSI 106 will determine what node (if any) is at fault from data received previously and will take action to downgrade the faulted node if necessary. When NetSI determines that a downgrade is required for a node, NetSI calls the appropriate NetPM operation to perform the downgrade. If a change in active status is required, NetSI calls the appropriate NetPM operation to set the active status. After NetPM is called to perform the downgrade, NetSI notifies ConfigMgr that the status is changing for a particular node. This allows the host to be informed immediately that a node is being downgraded. NetSI then writes an entry to the network configuration report indicating the status change and reason for it. NetSI downgrades nodes to the legal service state based on the current state of the node.

Detailed Description Text - DETX (68):

Referring to FIGS. 7C and 7D, process interaction between node management and network management is shown. Constant monitor (ConMon) 132 is an instance of an **object** running on an application node 136. ConMon 132 detects a faulted process or a failed configurable element, it notifies a service management process program 134. Service management process 134 determines if the configurable element failure causes the process to fall below its threshold level. If it does not, the service management process 134 restarts the configurable element. However, if the configurable element does fall below its threshold level then service management process 134 generates a configurable element status change message and forwards the notification to NodeSI 130. NodeSI forwards the configurable element status change to NodePM 112. NodePM 112 determines whether the configurable status change affects the run level of the node, which could cause a downgrade of the node. If the node is to be removed, NodePM 112 provides instructions to service management process 134 to remove all of the configurable elements necessary to achieve the downgraded state. NodePM 134 notified the NetPM 104 of the node status change. NetPM 104 performs a calculation to determine if the node status change affects the processor service group and application status. NetPM's calculation also determines if an auto-action, such as removing a node from in-service to min-set and restoring it again, should be performed on the node. If the node is to be removed, then the node status change is forwarded from NetPM to ConfigMgr 108. ConfigMgr notifies host 140 of the state change for the node, processor service group, and application. These state changes can be displayed or printed in a report.

Detailed Description Text - DETX (72):
The Configuration management subsystem (**class** name: ConfigMgr) provides the control interface between the SCP Host and Server components. All operations that can be performed on the server network are defined in this interface. The Configuration Management subsystem implements the following features:

Detailed Description Text - DETX (137):
Instantiates the NodeMAP **object,** and after getting the configuration information on the minimum Configurable elements that need to be configured on each servers, it brings up the server node to a minimal operational state (OS-MIN). From this state the server node is allowed only a minimum set of functionality such as bringing the rest of the processes up. The configuration data provided in each node's NodeMAP determines the capabilities of each server node (server nodes withplatform managercapabilities versus server nodes with query processing capabilities).

Detailed Description Text - DETX (138):
Creates the NodePM server **object** to handle the NetPM requests to perform operations within the same server node.

Detailed Description Text - DETX (139):
Per NetPM request, NodePM (through operations provided by its server **object**) can perform the following operations:

Detailed Description Text - DETX (147):
The Node System Integrity subsystem (**class** name NodeSI) provides fault isolation and monitoring services within a single server node. All process failures are logged by this subsystem and forwarded to node Management for recovery action. Node System Integrity implements the following features:

Detailed Description Text - DETX (152):
Faults detected by Constant Monitor **object** on each process.

Detailed Description Text - DETX (157):
NodeSI monitors the disk utilization on each server node, the issues appropriate IPR when the total capacity used on a particular file system exceeds a certain threshold. NodeSI communication with other **objects** is handled via the DOME interface. NodeSI gets the list of all IPUs in the configuration from NodePM. An array is set up containing the following information from each IPU:

Detailed Description Text - DETX (192):
Each Configurable element process is required to instantiate the Constant Monitor **object,** in order to detect and report abnormal conditions/events

generating different signals on the process. Constant Monitor reports these conditions via NotifyFault( ) operation of NodeSI. In case of failure to communicate the fault to NodeSI, the Constant Monitor may HALT the node, depending on the options set at the time of its instantiation.

Detailed Description Text - DETX (208):

FIG. 10 outlines the messages protocol that is used between SM and a Configurable element. If a configurable element cannot for link a service management interface (SMI) **object** into it, service management can still start that configurable element, but many of the features that service management provides will not be available.

Detailed Description Text - DETX (210):

The event manager subsystem provides the ability for a users to generically issue event notification to one or more registered parties. Multiple Event::Manager **object** instances may exist in the system. A node level Event::Manager exists on all nodes. Other Event::Manager instances may also exist to provide the ability for interested parties to register for events that are special to a process. The eventmanagerimpl program provides an Event::Manager **object** instance for the mode that it is running on. Events that are relevant to a node get issued through that Event::Manager instance. Users interested in events on a particular node can bind to that nodes Event::Manager instance by using that nodes name as the Event::Manager name. Programs can also embed an Event::Manager **object** within their program. The IprMgrImpl program is an example of a program that does this. The IprMgrImpl has an Event::Manager named IprEventMgr. Users that wish to receive IPR events. Users that are interested in a particular event may register with a particular Event::Manager instance to receive that event through that Event::Manager instance. The Event::Manager does not persistently store the list of registered parties. If the Event::Manager tries to forward an event to a Event::Receiver that has gone away, that Event::Receiver is removed form the list.

Detailed Description Text - DETX (211):

FIG. 11 shows two examples of uses for Event::Manager 250 in the telecom platform system. The eventmanagerimpl 252 contains the node Event::Manager **object** instance 250. The NodePMMain telecom platform program 254 uses this Event::Manager 250 to issue an event when the node changes state. The application program 256 then creates an Event::Receiver **object** 268 and passed a CORBA **object** reference to the register call on the "Node123" Event::Manager 250, When NodePMMain 254 generates an event by calling notify on the "Node123" Event::Manager 250, that Event:Manager 250 will find all of the Event::Receiver **objects** 258 that have registered to receive this event. Seeing that the application program has registered for this event, the Event::Manager 250 will call the notify( ) method on that Event::Receiver **object** 258 which will cause the notify ( ) method to be invoked in the Application program 256. In the example above, the Application program 256 has also registered with the "IprEventMgr" Event::Manager 260 in the IprMgrImpl program 262. When NodePMMMain 254 uses the IprMgrImpl interface to issue an IPR, the IprMgrImpl program 262 does the lookup on that IPR and performs verification, and calls

notify ( ) on the "IprEventMgr" Event::Manager 260. This cause that Event::Manager 250 to forward the generated event to the Event::Receiver 264 in the application program 256 that was passed in the register call.

Detailed Description Text - DETX (216):

Referring to FIG. 12, the IprMgrImpl program is the collection point for all IPRs in a telecom platform site. This program contains the IprMgrImpl CORBA server **object**. The IprMgrImpl **object** runs on each of the active/standby platform manager nodes. The active/standby state that the IprMgrImpl reacts to is the node level active/standby state of the telecom platform manager nodes. The standby IprMgrImpl **object** will unpublish its interface, and the active IprMgrImpl **object** will publish its CORBA interface when the platform manager nodes change active/standby state. By doing this, client users of both the IprMgr and IPRClient interfaces will have their IPRs forwarded to the active IprMgrImpl **object**.

Detailed Description Text - DETX (219):

The IPR subsystem has a two external interfaces: the IPRClient interface 274 and the CORBA IPR interface 276. The IPRClient interface 276 exists for backward compatibility with previous PAConfigurable element releases. Once the issued IPR from the IPRClient interface 274 has been converted by the IPRClient code, an IPR is issued using the IprMgrImpl CORBA interface to route the IPR to the active IprMgrImpl **object**. This interface still uses the LOCIPRDB.DSK IPR dictionary as input for converting the old PAConfigurable element IPRs to the current IPR subsystem format. This requires that a LOCIPRDB.DSK reside on each node that has programs that issue IPRs. The LOCIPRDB.DSK dictionary was used in the previous releases to do IPR verification before IPRs were forwarded to the host. The RegisterIPR utility is used to enter IPRs into the LOCIPRDB.DSK dictionary. The fields in the database entries include: ASCII key (IPR text), host IPR number, IPR priority, number of data words used, and data word format. In order to test the IPRMgr, IPRs must be defined in ipr.in which will be converted to a keyed dictionary (via the RegisterIPR utility).

Detailed Description Text - DETX (220):

The IprMgrImpl interface is a CORBA IDL interface. If an IPR is issued using this interface, it is not required to be entered in the LOCIPRDB.DSK dictionary. When the IprMgrImpl **object** receives an issued IPR, it looks it up in its IPR dictionary and constructs an IPR event to be issued. The IPR event contains information that was passed from the client that issued the IPR, and information from the IPR dictionary. IPRs must be added to the IPR dictionary and the MegaHub host IPR dictionaries prior to issuance of an IPRs. The IprDriver tool is used to add IPRs to the IprMgrImpl IPR dictionary. The reformat and reformat2 scripts exists to assist in converting a VAX IPR file to a format that can be used with the IprDriver to populate the IprMgrImpl IPR dictionary.

Detailed Description Text - DETX (229):

Referring to FIG. 15, the data collection subsystem (DC) 298 provides the traffic measuring functionality for the application programs within a node.

These measurements are counts recorded by the PegCounter **class** and elapsed time recorded by the TimeMeter **class**. PegCounter 299 testing will indirectly test shared memory 300 and semaphores. Client processes 301 peg to shared memory 300, and data collection 298 collects from shared memory 300 and sends to DCMaster 302. Every 30 minutes, data collection 298 sends the DCMaster 302 (in the active platform manager node) the 30 minutes worth of peg counter slots 299 and then data collection zeros out those slots. The active platform manager node 304 updates the standby platform manager node 306.

Detailed Description Text - DETX (236):
The threshold counter subsystem may be implemented as an **object** request broker (ORB) distributed **object**, using the orbeline ORB implementation. Applications are connected via Orbeline to a server **object** resident in the platform manager nodes. The server reports counter threshold crossings to applications via distributed **object** messaging environment (DOME). The server **object** are created by the thresholds counter server process, TCServer. Each TCServer process also communicates via Orbeline with the TCServers on remote nodes so that counters can be synchronized across sites. The TCServer keeps all counters in persistent storage using the persistent dictionary supplied in the common services library as template **class** RepShmDict.

Detailed Description Text - DETX (238):
Referring to FIG. 18, the threshold counter subsystem 360 API hides the orbeline-specific portions of the implementation from the application programmer. Instead, the client side of the subsystem will consist of two layers: an ORB-independent layer 362, and an orbeline-dependent layer 364. Although the orbeline-specific implementation of the subsystem is hidden from the application programmer, the distributed nature of the subsystem is not. To minimize the time required for counter increments, counter increments are buffered in the API, and sent to the server in batches. This means that the application is unable to receive immediate notification of the success or failure of some operations on the API **objects**.

Detailed Description Text - DETX (241):
As shown in FIGS. 19 and 20, the Message Handling subsystem 370 provides message based interprocessor communications services. Generally all interprocess communication between processes on the server nodes is carried out via the Distributed **Object** Messaging Environment (DOME) 372 shown in FIG. 21. DOME 372 uses the Message Handling subsystem 370 when information must be communicated across node boundaries. The Message Handling subsystem 370 is also used for communication to non-server external systems such as the SCP Host. The Message Handling subsystem 370 implements the following features.

Detailed Description Text - DETX (250):
Distributed **Object** Services

Detailed Description Text - DETX (251):
Referring to FIG. 21, DOME 372 is a client/server interface used for

interprocess client/server communication. It contains server interfaces 382 which allow server processes 382 to register **objects** and member functions for use by client processes 384. DOME 372 contains a shared memory database 380 to store the server descriptions and a stand-alone DOMEServices process (domeSrv) which maintains the server **object** descriptions from other nodes. It also contains client interfaces 384 which provide access to any registered server **object** in the node's DOME database.

Detailed Description Text - DETX (252):
The Interprocess Communications subsystem consists mainly of DOME. DOME provides the ability for a process to register a server **object** and it's methods in a way that allows other processes in the system to invoke those methods. DOME supports various modes of registration and access including many special routing options that aid in the development of fault resilient software. Features implemented by the Interprocess Communications subsystem include:

Detailed Description Text - DETX (253):
Registered **Object** Name Management across nodes and sites

Detailed Description Text - DETX (255):
Active/Standby **Object** request routing

Detailed Description Text - DETX (256):
Load Shared **Object** request routing

Detailed Description Text - DETX (257):
Broadcast **Object** request routing

Detailed Description Text - DETX (258):
Blocking/Non-Blocking **Object** requests

Detailed Description Text - DETX (261):
Command Line **Object**

Detailed Description Text - DETX (262):
Trace **Object**

Detailed Description Text - DETX (263):
Shared Memory **Object**

Detailed Description Text - DETX (264):
Semaphore **Object**

Detailed Description Text - DETX (265):
Keyed Dictionary **Object**

Detailed Description Text - DETX (266):
List **Object**

Detailed Description Text - DETX (267):
Replicated Keyed Dictionary **Object**

Detailed Description Text - DETX (268):
Shared Memory Dictionary **Object**

Detailed Description Text - DETX (270):
DbgTrace **Object**

Detailed Description Text - DETX (272):
The DbgCntl interface 404 is the control interface for DbgTrace **objects** 400. It allows users to specify many different aspects of the DbgTrace facility 400. This interface allows users to do the following things on DbgTrace **objects** 400:

Detailed Description Text - DETX (280):
The DbgTrace facility 400 allows the users to create different DbgTrace **objects** 400 that can each belong to one of multiple groups. This allows users to have a unique mask value for each group. All traces issued through the DbgTrace interface 400 get stored in an internal message buffer. Users can also specify whether to issue traces to standard error in addition to the internal buffer.

Detailed Description Text - DETX (281):
Trace **Object**

Detailed Description Text - DETX (282):
The Trace **object** provides the user the ability to optionally issue trace messages to standard error. When the user issues a trace, a mask is specified which represents the trace level that this trace will be output for. The Trace interface allows the user to specify a mask which all instances of trace in that UNIX process will use to determine whether or not to issue the trace message. The trace mask may supports eight unique mask values.

Detailed Description Text - DETX (284):
Referring to FIG. 23, Dictionary Management provides **classes** which are designed to support data storage and access. Dictionaries can be stored on disk (persistent) or stored in memory. Dictionaries can also be private (used

by local process only) or shared (accessible by multiple processes). The purposes of these dictionaries are defined by the application program. The primary interaction between DmsMaster 430 and DmsServer 432 is that DmsMaster 430 updates DmsServer 432 when it receives an update message from the application. DmsMaster 430 runs as active/standby in the platform manager nodes, and DmsServer 432 runs in all (or a subset) of the IPUs.

Detailed Description Text - DETX (286):
Event services provide the capability to generate and distribute specific occurrences significant to a task among loosely coupled processes. An example of an event is the completion of an input/output transfer. The event services may be a CORBA-based interprocess communication facility. It uses standard CORBA requests that result in the execution of an operation by an **object**. This is accomplished through the event manager implemementation program.

Detailed Description Text - DETX (287):
By defining two distinct roles for **objects,** communication is decoupled between **objects**; creating asynchronous communication. One **object** receives and accumulates new events, while the other **object** registers an interest to be forwarded these new events. This is accomplished by two CORBA **classes,** EventManager and EventReceiver. EventManager provides an interface definition language (IDL) interface for receiving new events. EventReceiver provides an interface definition language interface for clients interested in receiving events.

Claims Text - CLTX (7):
supplying distributed **object** processes operable to provide a distributed database repository for **object**-based communications.

Claims Text - CLTX (63):
distributed **object** processes operable to provide a distributed database repository for **object**-based communications.

US-PAT-NO: **6574664**

DOCUMENT-IDENTIFIER: US 6574664 B1

TITLE: Apparatus and method for IP and MAC address discovery at the process layer

---------- KWIC ---------

US Patent No. - PN (1):
6574664

Application Filing Date - AD (1):
19990129

Brief Summary Text - BSTX (7):
Simple Network Management Protocol (SNMP) is a standard for network management that can be employed by the network manager to manage the devices on the network. Each SNMP-manageable device has a Management Information Base (MIB) that stores **objects** or variables representing different characteristics of a device including its IP and MAC addresses. The network manager discovers the identity of a device by sending a GetNext request to a SNMP-manageable device. The response returns the IP and/or MAC addresses.

Drawing Description Text - DRTX (2):
For a better understanding of the nature and **objects** of the invention, reference should be made to the following detailed description taken in conjunction with the accompanying drawings, in which:

Detailed Description Text - DETX (8):
The ARP layer is used by the IP layer to translate an IP address into its respective MAC address. The MAC address is then used in the delivery of the data packet to the intended device. The ARP layer performs the address resolution through dynamic binding. When a device A want to resolve the IP address of device B, IP.sub.B, a broadcast message is sent on the network with IP address IP.sub.B. Device B recognizes its IP address and responds with the physical address. An ARP cache is used to store those recently translated IP and MAC addresses in order to minimize the number of address translations that are performed at the ARP layer. However, the MAC addresses stored in the ARP cache cannot be **accessed by an application program at the process layer**.

Detailed Description Text - DETX (11):
The local discovery node 132, 136 uses SNMP to discover the devices on the network. SMNP is an application program that provides multi-vendor,

interoperable network management. Each SNMP device is associated with a MIB that stores **objects** and/or variables representing different characteristics of a resource. The MIB is composed of a number of tables that are scattered through out the various network protocol layers. The MIB includes an "atTable" which is in essence, the ARP cache. An exemplary atTable is shown below as Table 1.

Detailed Description Text - DETX (18):

The local IP and MAC discovery procedure 174 needs to determine which IP addresses are active. In the case of a **class** C network, such as an ethernet LAN, the network can support up to 255 IP addresses, but not all of the 255 IP addresses may be active. A **class** A network can support up to 2.sup.24 or 16,777,216 IP addresses. Furthermore, the atTable 180 has room for a limited number of entries and as such, is refreshed on a periodic basis in order to store the most recently used entries. Due to the large number of IP addresses available and the space limitation of the atTable 180, the procedure 174 executes bursts of IP addresses at a time. A burst is a subset or a predefined number of the IP addresses supported on a network. By operating on a select number of IP addresses at one time, the procedure 174 can obtain the corresponding MAC addresses before they are deleted from the atTable 180.

US-PAT-NO: **6581088**

DOCUMENT-IDENTIFIER: US 6581088 B1

TITLE: Smart stub or enterprise javaTM bean in a distributed processing system

---------- KWIC ---------

US Patent No. - PN (1):
6581088

Application Filing Date - AD (1):
19990923

Brief Summary Text - BSTX (8):
A variety of different types of software may be used to program application server 103 and/or client 105. One programming language is the Java.TM. programming language. Java.TM. application **object** code is loaded into a Java.TM. virtual machine ("JVM"). A JVM is a program loaded onto a processing device which emulates a particular machine or processing device. More information on the Java.TM. programming language may be obtained at http://www.javasoft.com, which is incorporated by reference herein.

Brief Summary Text - BSTX (10):
RMI 100a is a distributed programming model often used in peer-to-peer architecture described below. In particular, a set of **classes** and interfaces enables one Java.TM. **object** to call the public method of another Java.TM. **object** running on a different JVM.

Brief Summary Text - BSTX (16):
FIG. 2 illustrates peer-to-peer architecture 214. Processing devices 216, 217 and 218 are coupled to communication medium 213. Processing devices 216, 217, and 218 include network software 210a, 210b, and 210c for communicating over medium 213. Typically, each processing device in a peer-to-peer architecture has similar processing capabilities and applications. Examples of peer-to-peer program models include Common **Object** Request Broker Architecture ("CORBA") and Distributed **Object** Component Model ("DCOM") architecture.

Brief Summary Text - BSTX (31):
According to another aspect of the present invention, the first processing device includes an Enterprise Java.TM. Bean **object**.

Brief Summary Text - BSTX (46):
According to another aspect of the present invention, the article of manufacture comprises a first set of digital information, including an Enterprise Java.TM. Bean **object** for selecting a service provider from a plurality of service providers.

Drawing Description Text - DRTX (12):
FIG. 5b illustrates an EJB **object** architecture;

Detailed Description Text - DETX (4):
FIG. 3a illustrates a simplified block diagram 380 of the software **layers in a processing** device of a clustered enterprise Java.TM. system, according to an embodiment of the present invention. A detailed description of a clustered enterprise Java.TM. distributed processing system is described below. The first layer of software includes a communication medium software driver 351 for transferring and receiving information on a communication medium, such as an ethernet local area network. An operating system 310 including a transmission control protocol ("TCP") software component 353 and internet protocol ("IP") software component 352 are upper software layers for retrieving and sending packages or blocks of information in a particular format. An "upper" software **layer is generally defined as a software component which utilizes or accesses one or more "lower" software layers** or software components. A JVM 354 is then implemented. A kernel 355 having a remote Java.TM. virtual machine 356 is then layered on JVM 354. Kernel 355. described in detail below, is used to transfer messages between processing devices in a clustered enterprise Java.TM. distributed processing system. Remote method invocation 357 and enterprise Java.TM. bean 358 are upper software layers of kernel 355. EJB 358 is a container for a variety of Java.TM. applications.

Detailed Description Text - DETX (12):
In particular, server 302, server 303, and client 304 have kernels 302b, 303b, and 304b, respectively. In particular, in order for two JVMs to interact, whether they are clients or servers, each JVM constructs an RJVM representing the other. Messages are sent from the upper layer on one side, through a corresponding RJVM, across the communication medium, through the peer RJVM, and delivered to the upper layer on the other side. In various embodiments, messages can be transferred using a variety of different protocols, including, but not limited to, Transmission Control Protocol/Internet Protocol ("TCP/IP"), Secure Sockets Layer ("SSL"), Hypertext Transport Protocol ("HTTP") tunneling, and Internet InterORB Protocol ("IIOP") tunneling, and combinations thereof. The RJVMs and socket managers create and maintain the sockets underlying these protocols and share them between all **objects** in the upper layers. A socket is a logical location representing a terminal between processing devices in a distributed processing system. The kernel maintains a pool of execute threads and thread manager software component 364 multiplexes the threads between socket reading and request execution. A thread is a sequence of executing program code segments or functions.

Detailed Description Text - DETX (25):
The first line establishes a session with the acme server using the t3 protocol. If RJVMs do not already exist, each JVM constructs an RJVM for the other and an underlying TCP socket is established. The client-side representation of this session--the T3Client **object**--and the server-side representation communicate through these RJVMs. The server-side supports a variety of services, including database access, remote file access, workspaces, events, and logging. The second line obtains a LogServices **object** and the third line writes the message.

Detailed Description Text - DETX (26):
Clustered enterprise Java.TM. computer architecture 300 also supports a server-neutral syntax consistent with a peer-to-peer distributed processing architecture. As an example, the following code fragment obtains a stub for an RMI **object** from the JNDI-compliant naming service on a server and invokes one of its methods. Hashtable env=new Hashtable( ); env.put(Context.PROVIDER_URL, "t3://acme:7001"); env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WebLogicinitialContextFactory"); Context ctx=new InitialContext(env); Example e=(Example) ctx.lookup("acme.eng.example"); result=e.example(37);

Detailed Description Text - DETX (27):
In an embodiment, JNDI naming contexts are packaged as RMI **objects** to implement remote access. Thus, the above code illustrates a kind of RMI bootstrapping. The first four lines obtain an RMI stub for the initial context on the acme server. If RJVMs do not already exist, each side constructs an RJVM for the other and an underlying TCP socket for the t3 protocol is established. The caller-side **object**--the RMI stub--and the callee-side **object**--an RMI impl-communicate through the RJVMs. The fifth line looks up another RMI **object,** an Example, at the name acme.eng.example and the sixth line invokes one of the Example methods. In an embodiment, the Example impl is not on the same processing device as the naming service. In another embodiment, the Example impl is on a client. Invocation of the Example **object** leads to the creation of the appropriate RJVMs if they do not already exist. II. Replica-Aware or Smart Stubs/EJB **Objects**

Detailed Description Text - DETX (28):
In FIG. 3c, a processing device is able to provide a service to other processing devices in architecture 300 by replicating RMI and/or EJB **objects**. Thus, architecture 300 is easily scalable and fault tolerant. An additional service may easily be added to architecture 300 by adding replicated RMI and/or EJB **objects** to an existing processing device or newly added processing device. Moreover, because the RMI and/or EJB **objects** can be replicated throughout architecture 300, a single processing device, multiple processing devices, and/or a communication medium may fail and still not render architecture 300 inoperable or significantly degraded.

Detailed Description Text - DETX (30):
RA RMI stub 580 is a Smart stub which is able to find out about all of the

service providers and switch between them based on a load balancing method 507 and/or failover method 508. In an embodiment, an RA stub 580 includes a replica handler 506 that selects an appropriate load balancing method 507 and/or failover method 507. In an alternate embodiment, a single load balancing method and/or single failover method is implemented. In alternate embodiments, replica handler 506 may include multiple load balancing methods and/or multiple failover methods and combinations thereof. In an embodiment, a replica handler 506 implements the following interface: public interface ReplicaHandler [**Object** loadBalance(**Object** currentProvider) throws RefreshAbortedException; **Object** failOver(**Object** failedProvider, RemoteException e) throws RemoteException;]

Detailed Description Text - DETX (49):

A RMI compiler recognizes a special flag that instructs the compiler to generate an RA stub for an **object**. An additional flag can be used to specify that the service methods are idempotent. In an embodiment, RA stub 580 will use the replica handler described above and illustrated in FIG. 5a. An additional flag may be used to specify a different handler. In addition, at the point a service is deployed, i.e., bound into a clustered naming service as described below, the handler may be overridden.

Detailed Description Text - DETX (50):

FIG. 5b illustrates another embodiment of the present invention in which an EJB **object** 551 is used instead of a stub, as shown in FIG. 5a. III. Replicated JNDI-compliant Naming Service

Detailed Description Text - DETX (58):

In this example, the two calls to example may be handled by different service providers since the Smart stub is able to switch between them in the interests of load balancing. Thus, the Example service **object** cannot internally store information on behalf of the application. Typically the stateless model is used only if the provider is stateless. As an example, a pure stateless provider might compute some mathematical function of its arguments and return the result. Stateless providers may store information on their own behalf, such as for accounting purposes. More importantly, stateless providers may access an underlying persistent storage device and load application state into memory on an as-needed basis. For example, in order for example to return the running sum of all values passed to it as arguments, example might read the previous sum from a database, add in its current argument, write the new value out, and then return it. This stateless service model promotes scalability.

Detailed Description Text - DETX (63):

In the stateful programming model, a service provider is a long-lived, stateful **object** identified by some unique system-wide key. Examples of "entities" that might be accessed using this model include remote file systems and rows in a database table. A targeted provider may be accessed many times by many clients, unlike the other two models where each provider is used once by one client. Stubs for targeted providers can be obtained either by direct

lookup, where the key is simply the naming-service name, or through a factory, where the key includes arguments to the create operation. In either case, the stub will not do load balancing or failover. Retries, if any, must explicitly obtain the stub again.

Detailed Description Text - DETX (64):

There are three kinds of beans in EJB, each of which maps to one of the three programming models. Stateless session beans are created on behalf of a particular caller, but maintain no internal state between calls. Stateless session beans map to the stateless model. Stateful session beans are created on behalf of a particular caller and maintain internal state between calls. Stateful session beans map to the stateless factory model. Entity beans are singular, stateful **objects** identified by a system-wide key. Entity beans map to the stateful model. All three types of beans are created by a factory called an EJB home. In an embodiment, both EJB homes and the beans they create are referenced using RMI. In an architecture as illustrated in FIGS. 3-5, stubs for an EJB home are Smart stubs. Stubs for stateless session beans are Smart stubs, while stubs for stateful session beans and entity beans are not. The replica handler to use for an EJB-based service can be specified in its deployment descriptor.

Detailed Description Text - DETX (65):

To create an indirect RMI-based service, which is required if the **object** is to maintain state on behalf of the caller, the application code must explicitly construct the factory. A targeted RMI-based service can be created by running the RMI compiler without any special flags and then binding the resulting service into the replicated naming tree. A stub for the **object** will be bound directly into each instance of the naming tree and no service pool will be created. This provides a targeted service where the key is the naming-service name. In an embodiment, this is used to create remote file systems. V. Hardware and Software Components

Claims Text - CLTX (8):

8. An article of manufacture including an information storage medium wherein is stored information, comprising: a first set of digital information, including a Java.TM. virtual machine with a Java.TM. bean **object** for selecting a service provider from a plurality of service providers; wherein the Java.TM. bean **object** has a load balancing software component that selects a particular service provider, from the plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and, wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in a transaction between either of the service provider or server and the client processing device.

Claims Text - CLTX (9):

9. The article of manufacture of claim 8, wherein the Java.TM. bean **object** has a failover software component for removing a failed service provider from a

list of service providers.

US-PAT-NO: **6185590**

DOCUMENT-IDENTIFIER: US 6185590 B1

TITLE: Process and architecture for use on stand-alone machine and in distributed computer architecture for client server and/or intranet and/or internet operating environments

--------- KWIC ---------

US Patent No. - PN (1):
6185590

Application Filing Date - AD (1):
19971015

Brief Summary Text - BSTX (44):
The present invention is also based, in part, on my discovery that the **object** manager and engine **object** component layers may be advantageously be designed to operate independently, thereby making possible a distributed computing environment, as described below in detail. I have further discovered that an efficient method of implementing the engine **object** component layer is by using pre-populated tables/files. I have further discovered that the engine management layer may be advantageously divided into a three layer structure of load/unload engine, dynamic linking engine function calls, and initialize engine setting.

Brief Summary Text - BSTX (45):
In accordance with one embodiment of the invention, a computer implemented process migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an **object** for each engine. The **object** provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented process includes the step of providing an engine management function interfacing with the program specific API. The engine management function furnishes a protective wrapper for each function call associated with the engine, trapping errors, and provides error management and administration to prevent conditions associated with improper engine functioning. The process optionally includes the step of providing an engine configuration function transforming API calls received from the program specific API into standardized calls. The engine configuration function provides additional functionality, including safely loading and unloading the engine. The process optionally includes the step of providing an engine function managing the standardized calls for each engine, thereby providing substantially uniform access to the engine and the engine settings associated with the engine.

Brief Summary Text - BSTX (46):

In accordance with another embodiment of the invention, a computer implemented method migrates at least one program specific Application Programmer Interface (API) from an original state into a generic interface by building an **object** for each engine. The **object** provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented method includes the steps of defining a substantially consistent interface for individual **object** components that represent diverse technologies, and migrating a plurality of engines to the consistent interface. The computer implemented method also includes the step of substantially automatically and/or substantially uniformly, managing the individual **object** components using a predefined **object** manager and the consistent interface.

Brief Summary Text - BSTX (47):

In accordance with another embodiment of the invention, a computer architecture migrates at least one program specific Application Programmer Interface (API) from an original state into a generic interface by building an **object** for each engine. The **object** provides substantially uniform access to the engine and engine settings associated with the engine. The computer architecture includes an engine management layer interfacing with the program specific API and providing engine management and administration, an engine configuration layer transforming API calls received from the program specific API into standardized calls, and an engine layer managing the standardized calls for each engine.

Brief Summary Text - BSTX (48):

In accordance with another embodiment of the invention, an engine management layer configures a computer architecture to perform one or more computer implemented or computer assisted operations. The computer operations include one or more of loading and unloading engine dynamic link libraries into and out of memory for each engine, mapping at least one engine function to at least one corresponding engine **object,** providing general error detection and error correction for each engine, determining and matching arguments and returning values for mapping the at least one engine function to the at least one corresponding engine **object,** and/or managing error feedback from the at least one program specific API.

Brief Summary Text - BSTX (49):

In accordance with another embodiment of the invention, a distributed computer system migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an **object** for each engine. The **object** provides substantially uniform access to the engine and engine settings associated with the engine. The distributed computer system includes a server configured to include at least one engine having an engine interface providing one or more features to be executed, and at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine. The distributed computer system also includes at least one

client configured to be connectable to the server and optionally configured to be connectable to another server. The client includes an **object** manager layer communicable with and managing the at least one engine component stored on the server via the substantially consistent interface.

Brief Summary Text - BSTX (50):

In accordance with another embodiment of the invention, a distributed computer implemented process migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an **object** for each engine. The **object** provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented process includes the step of providing, on a server, at least one engine having an engine interface, and providing one or more features to be executed. The computer implemented process also includes the step of providing, on at least one of the server and another server connectable to the server, at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine. The computer implemented process also includes the step of providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an **object** manager layer communicable with and managing the at least one engine component via the substantially consistent interface.

Brief Summary Text - BSTX (53):

In accordance with another embodiment of the invention, a computer readable tangible medium is provided that stores an **object** thereon, for execution by the computer.

Brief Summary Text - BSTX (54):

These together with other **objects** and advantages which will be subsequently apparent, reside in the details of construction and operation as more fully herein described and claimed, with reference being had to the accompanying drawings forming a part hereof wherein like numerals refer to like elements throughout.

Drawing Description Text - DRTX (5):

FIG. 4 is an illustration of the design of an **Object** in accordance with the computer architecture of the present invention;

Detailed Description Text - DETX (29):

A simplified overview of the architecture is illustrated in FIG. 3. In FIG. 3, the component interface 8 sits on top of an **Object** Manager 14 that communicates with individual **objects** e.g., 16, 18, 20. These **objects** 16, 18, 20 represent specific core technologies that are represented as "C"-level APIs. The design of **Object1, Object2, . . . ObjectN** is illustrated in FIG. 4.

Detailed Description Text - DETX (35):

The architecture consists of a hierarchical series of layers that take any "C"-level API from its unique state to one that is standard and consistent. The result is a single, highly-integrated **object** component that contains and manages any type of engine that can be programmed regardless of the nature and subject of the core technology. The architecture therefore not only defines the goal (e.g., the **object** component interface) but also the means of implementing that goal for any type of engine.

Detailed Description Text - DETX (36):
The architecture is comprised of two major parts as illustrated in FIG. 5: the **Object** Manager 14, and the individual **object** components 16, 18, 20. The **Object** Manager 14 in FIG. 5 manages individual **object** components 16, 18, 20 illustrated as **Object** 1, **Object** 2, etc. The **Object** Manager 14 communicates with the individual **object** components 16, 18, 20 using a consistent COM interface.

Detailed Description Text - DETX (37):
Each **object** component implements the feature set of an individual engine by mapping a consistent COM interface to the "C"-Level API interface of the individual engine that it supports. In this way the **Object** Manager can consistently communicate with each engine, using the engine's **object** component. Because the COM interface of each **object** component is consistent, the **Object** Manager can interface with every underlying engine the same way.

Detailed Description Text - DETX (39):
1) definition of consistent COM interfaces for individual **object** components that represent diverse technologies;

Detailed Description Text - DETX (41):
3) a predefined **Object** Manager that automatically manages the individual **object** components.

Detailed Description Text - DETX (42):
When implemented, for example, as an ActiveX control, the architecture also yields an umbrella control that can be used by a high-level programmer to program and manage numerous sophisticated technologies in a plug-and-play environment. In order to facilitate the discussion of the architecture itself it is best to start with the architecture of the engine **object** component and then describe the **Object** Manager. Since the **Object** Manager is directly dependent on the engine **object** components, an understanding of the latter will assist in the description of the former.

Detailed Description Text - DETX (43):
Engine **Object** Component--16, 18, 20

Detailed Description Text - DETX (44):
The purpose of the engine **object** is to wrap a specific engine using a series

of layers that convert the engine's unique interface into a COM interface that is, for example, specified by the architecture. The architecture not only defines the consistent COM interface for implementing an engine, it also describes how to implement the interface from the original "C"-Level API. Once the COM interface of the engine **object** component is implemented, the **Object** Manager understands and can therefore communicate with it.

Detailed Description Text - DETX (45):
Each engine component consists of, for example, three layers that are designed to migrate the original API of the engine to a consistent COM interface. As illustrated in FIG. 6, the **Object** Manager 14 communicates with the topmost layer 22 of the **object** component 16, 18, 20 which is the defined interface of **object** component.

Detailed Description Text - DETX (46):
Each layer is described below in two parts. The first part is the prescribed COM interface for communicating with the engine **object** component. The second part describes a specific path for automating building the layer. By providing an outline for automating building each layer, the overall engine **object** component can be automatically, substantially automatically or manually expedited and generated.

Detailed Description Text - DETX (48):
The first layer in the **object** component architecture is designed to deal with the fundamental features of an engine. This includes the ability to load and unload the standard or commercially available via, for example, MicroSoft Corporation, engine Dynamic Link Libraries (DLLs) into memory, as well as the ability to consistently deal with errors. This is the most fundamental layer because it is the essential "wrapper" layer of an engine. Once this layer is complete all interaction with the underlying engine is filtered through this layer. Additional important engine management functions include dynamically accessing a function call of an engine, and initializing engine settings. All of these engine management functions are optionally and beneficially table driven to promote or facilitate access to, and implementation of, engine management functions.

Detailed Description Text - DETX (55):
The IEngineManagement interface is implemented in the C++ **class** as the public methods: ActivateEngine( ) and IsEngineActivated( ).

Detailed Description Text - DETX (56):
The first step of implementing the Engine Management layer 20 is to wrap each original engine function within a **class**-defined function that represents the original. For example, if there is an original function called Somefunction( ), then the engine **object** should have a corresponding Somefunction( ) method. The engine **object** version can then add standard engine and error management code so that any layers above have automatic error detection, correction, and reporting.

Detailed Description Text - DETX (58):
Given the original function name, the GetProcAddress can map the original function to one that is defined by the engine **object**. Using the engine **object** C++ header file described above, the Somefunction( ) method is mapped to the original engine function using the following line of code:

Detailed Description Text - DETX (59):
To map all the function calls within the original engine DLLs just requires cycling through each function call and mapping it to the engine **object** counterpart. Since Windows contains facilities that enables access to all the functions within a DLL, a simple loop may be used. The hLib module is derived from the DLL name, which, as mentioned at the start, is the one piece of information we are given.

Detailed Description Text - DETX (60):
What is more complex is to define a generic implementation of the engine **object** version of the original function. This may be described in code as follows:

Detailed Description Text - DETX (61):
The engine **object** version of the original function passes the function call to the original one after completing a series of assertion tests, and is followed by a series of error detection tests. In this way the original engine function is "wrapped" by the engine **object** to manage error detection and correction.

Detailed Description Text - DETX (63):
The LoadDLLs function is a generic implementation of a function that loops through the names of DLLs that are provided (in the form of the m_Modules variable), and cycles through each one loading it into memory using the Windows LoadLibrary( ) function. A similar engine **object** function can be implemented to remove these DLLs from memory.

Detailed Description Text - DETX (69):
Mapping original functions to engine **object** counterparts

Detailed Description Text - DETX (71):
Determining and matching arguments and return values for mapping the original functions to their engine **object** counterparts. In order to add assertion and error detection and correction, the original function must be wrapped and called from within the engine **object** version of the original function.

Detailed Description Text - DETX (72):

Managing error feedback. All APIs have their own way of providing error feedback. Since one of the goals of the Engine Management layer is to generically manage error detection, correction, and feedback, it must handle all errors identically. However, APIs have numerous and incompatible methods in this case. I have determined that most APIs follow one of several distinct mechanisms for providing error feedback. By creating specific **classes** of APIs, the process of generating Layer 1 engine management may be expedited, manually and/or automatically.

Detailed Description Text - DETX (74):
The second layer 24 in the **object** component architecture is designed to deal with configuring an engine. This includes the ability to set any variety of features that are generally associated with the functioning of an engine. The architecture is designed to meet the challenge of providing a uniform interface for dealing with generally any or most engine settings.

Detailed Description Text - DETX (75):
The engine configuration layer 24 includes a series of prefabricated functions that map out the settings stored in the table to the appropriate engine configuration parameters. Accordingly, all that is needed is to fill in the values for the table associated with engine configuration. Thus, the engine **object** may advantageously come pre-packaged with predetermined tables populated with predetermined values.

Detailed Description Text - DETX (86):
The third layer 22 in the **object** component architecture is designed to deal with accessing the actual functionality of the core engine. For example, for an OCR engine this would be to OCR an image or a document. For a text retrieval engine this would be to initiate and retrieve results of a text search.

Detailed Description Text - DETX (121):
On top of these, a three-level C++ **class (or object)** 102 is built for each engine. This **object** gives uniform access to the engine and to all its unique settings. The three levels do the following:

Detailed Description Text - DETX (122):
Level 1 of the C++ **classes** 112 is a protective wrapper for each function call in the underlying engine. It traps all errors and provides error management and administration to prevent accidental GPFs or engine crashes.

Detailed Description Text - DETX (123):
Think of it as the "condom layer." While providing the most direct access feasible to the underlying engine and all its capabilities, level 1 of the C++ **class** 112 also protects the user from the engine. It manages all engine loading and unloading, prevents multiple copies of an engine and calls engines automatically as needed.

Detailed Description Text - DETX (125):
Level 2 of the C++ **classes** 114 bridges the low-level API calls so they can be used by level 3 116 in standardized calls for each category. And it supplements the engine by providing additional functionality, such as safely loading and unloading engines.

Detailed Description Text - DETX (126):
Level 3 of the C++ **class** 116 consists of a standardized set of calls for all engines in each category. Programmers can access all the unique functions of each engine in a uniform way.

Detailed Description Text - DETX (127):
Another associated C++ **class,** called a Visual **Class** 104, adds a visual interpretation of each engine. This **class** manages all user interaction with each underlying engine. Like their lower-level counterparts, the Visual **Class** consists of three layers:

Detailed Description Text - DETX (130):
Level 3--122 manages anything else from the underlying engine (such as annotations) that needs to appear on the window. The Visual **Class** includes engine-specific Windows dialog boxes that let you customize which engine features you want to use, as well as any other Windows representation necessary for a toolkit. (For example, a compression engine has to display the image--the visual **class,** not the engine, does the work.)

Detailed Description Text - DETX (131):
The **Object** Manager layer 106, the first horizontal umbrella, orchestrates the underlying **objects**. It translates service requests into a form that the engine **objects** can understand.

Detailed Description Text - DETX (132):
The Windows Manager 108 presents Windows messages (move window, mouse/scrollbar/toolbox activity) to the **Object** Manager. It is written using Microsoft's Foundation **Class** (MFC), which makes it easy to support OCXs. (The OCX is in fact an MFC **class.**)

Detailed Description Text - DETX (134):
Accordingly, the present invention provides two main layers, the engine **object** component layer and the **object** manager layer. By creating these two main layers, the present invention allows third parties to create their own engine **object** component layers so that the third party engine can be readily compatible and useable by the present invention. In addition, the present invention is accessible via the Internet. That is, the present invention is operable over the Internet using, for example, standard Internet protocols, such as component **object** module (COM) communication protocol and distributed

COM (DCOM) protocol.

Detailed Description Text - DETX (135):
In addition, the present invention optionally combines three layers of functions including the visual interface, the windows manager and the **object** manager into one layer called the **object** manager. Of course, this combination of layers is not meant to convey that only these specific layers must be used, but rather, to be indicative of overall functionality generally required to implement or execute component engines. That is, one or more of the above functions may be incorporated into the **object** manager layer. The present invention also advantageously combines the visual **classes** and C++ **classes** into the engine **object** component to further standardize and/or provide access to the **object** manager for engine **object** components.

Detailed Description Text - DETX (136):
The present invention optionally uses the standard ActiveX component control supplied, for example, by MicroSoft Corporation. ActiveX is a protocol for component communication. The present invention also creates each of the **object** manager and the engine component layer as a separate ActiveX. That is, the **object** manager is its own ActiveX control, and the engine **object** is its own ActiveX control. Thus, the engine **object** can now run independently from the **object** manager. Accordingly, the engine **object** can operate without relying necessarily on the concurrent operation of the **object** manager.

Detailed Description Text - DETX (137):
The independent relationship between the engine **object and the object** manager means also that the engine **object** represents a discrete means of technology. For example, an engine **object** can be an OCR technology. This provides several benefits. First, because the **object** manager layer is open, the manufacturer of the OCR technology can wrap their own engine in the form of an engine **object** component, and the engine will automatically "plug into" or work with, the **object** manager. Thus, the engine **object** is provided high level access, making it accessible to many more parties, users, and the like. When the **object** manager interface is designed to be open, any third party, such as an engine manufacturer, can create their own engine **object** component that is compatible with the **object** manager, the manufacturer can do it.

Detailed Description Text - DETX (138):
FIG. 19 is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for client server and/or intranet operating environments. A very significant point that is relevant to why the **object** manager and the engine **object** component are independent in the present invention relates to providing a distributed environment for using the present invention. Rather than communicate within the same technology between the **object** manager and the engine **object, the object** manager and the engine **object** component communicate with each other in binary mode, via, for example, standard distributed component **object** module (DCOM) communication. As illustrated in FIG. 19,

object manager 14 communicates with engine object component 16, 18, 20 via DCOM specification 166. Other types of component communication may also be utilized that provide the capability of a distributed component interaction.

Detailed Description Text - DETX (139):
Thus, the engine object component and the object manager can leverage current protocols to not only communicate on the same machine, but also on different machines such as a client server and/or intranet and/or Internet environment. The object manager can be placed on one machine, and the engine object component on another machine and have distributed processing, what is otherwise called thin client processing, distributed processing, wide area intranet processing.

Detailed Description Text - DETX (140):
What this allows the present invention to do is to put the object manager on the thin client, who would accept the request from the user, for example, to OCR something or to print something. The actual request is handled or processed by the engine object component which generally resides on the server. The engine object component contains the horse power, or the processing power to process the request.

Detailed Description Text - DETX (141):
The engine object layer is generally located in the same or substantially same location as where the core technology or engine itself is being stored. Alternatively, the engine object layer and the engine may be optionally located in a distributed environment on different machines, servers, and the like.

Detailed Description Text - DETX (142):
FIG. 20 is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for client server and/or intranet operating environments. In FIG. 20, client 170 includes object manager layer 172. Client 170 executes the core technology 180, via accessing engine object layer 178 managed/stored on server 176, and communicated via server 174.

Detailed Description Text - DETX (143):
Client 182, located on the same server 176 as core technology 180 and engine object layer 178, may also be used to execute the core technology 180 via object manager layer 184. In this instance, the client 182 with the object manager layer 184 is located on the same server 176 as the engine object layer 178. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine object component layer and the object manager layer on the same machine 186, as well as the core technology.

Detailed Description Text - DETX (144):
Further, since the object manager is formatted or constructed of a client

technology, the **object** manager can sit in a standard browser. This means that anyone that has an Internet browser, i.e., anyone that has access to the world wide web (WEB) can actually access the core engine technology. Thus, by structuring the architecture of the present invention as described herein, users automatically become Internet, intranet and/or WEB enabled.

Detailed Description Text - DETX (145):
The present invention also transforms the core technology from essentially client based technology into a client server and/or a thin client technology. This makes the core technology high level accessible, thereby transforming any core technology into client server, or hidden client technology. The browser is located on the client, and the browser leverages the **object** manager. Accordingly, the browser optionally contains the **object** manager, and the **object** manager makes requests over, for example, the Internet, local network, and the like via a server, to the engine **object**. The server would be either a web server or a LAN server.

Detailed Description Text - DETX (146):
The present invention also advantageously provides the ability to have the client and the server, in a distributed environment as discussed above, or on the same machine locally. The present invention utilizes the DCOM communication protocol defining the communication protocol between the **object** manager and the engine **object** component. Accordingly, since DCOM can work on the same machine as well as in a distributed environment, DCOM does not necessitate that the engine **object or the object** manager component be on two separate machines.

Detailed Description Text - DETX (147):
FIG. 21 is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for network environments, such as the Internet. As illustrated in FIG. 21, **object** manager 14 communicates with engine **object** component 16, 18, 20 via DCOM specification and a networking environment, such as the Internet, intranet, and the like 168. Other types of component communication may also be utilized that provide the capability of a distributed component interaction over a networking environment.

Detailed Description Text - DETX (148):
FIG. 22 is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components in the Internet environment. In FIG. 22, client 170 includes **object** manager layer 172. Browser/thin client 170 a executes the core technology 180, via accessing engine **object** layer 178 managed/stored on web server 176a, and communicated via the Internet 174a.

Detailed Description Text - DETX (149):
Browser/thin client 182a, located on the same web server 176a as core technology 180 and engine **object** layer 178, may also be used to execute the

core technology 180 via **object** manager layer 184. In this instance, the browser/thin client 182a with the **object** manager layer 184 is located on the same web server 176a as the engine **object** layer 178. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine **object** component layer and the **object** manager layer on the same machine 186, as well as the core technology.

Detailed Description Text - DETX (154):

FIG. 24 is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments. The architecture in FIG. 24 provides the capability to perform the viewer process off-line. That is, the viewer process 188 provides an added feature on top of the **object** manager layer 14. As described above, **object** manager layer 14 is essentially an interface, and the viewer process 188 is an application that leverages the **object** manager layer 14.

Detailed Description Text - DETX (155):

The advantage of the viewer process 188 being built on the **object** manager layer 14, which is built on top of the engine **object** layer 16, 18, 20, is that the viewer process can offset its processing capabilities anywhere in a distributed environment. It can have the processing occur at the local station, on a server, and the like, as described below in detail. Significantly, the **object** manager and the engine **object** component are independent to provide a distributed environment for using the present invention. Rather than communicate within the same technology between the **object** manager and the engine **object, the object** manager and the engine **object** component communicate with each other in binary mode, via, for example, standard distributed component **object** module (DCOM) communication.

Detailed Description Text - DETX (156):

As illustrated in FIG. 24, **object** manager 14 communicates with engine **object** component 16, 18, 20 via DCOM specification 166. Other types of component communication may also be utilized that provide the capability of a distributed component interaction. **Object** manager 14 is also respectively connectable to viewer process 188.

Detailed Description Text - DETX (157):

Thus, the engine **object** component and the **object** manager can leverage current protocols to not only communicate on the same machine, but also on different machines such as a client server and/or intranet and/or Internet environment. The **object** manager and/or viewer process can be placed on one machine, and the engine **object** component on another machine and have distributed processing, what is otherwise called thin client processing, distributed processing, wide area intranet processing.

Detailed Description Text - DETX (158):

What this allows the present invention to do is to put the **object** manager on

the thin client, who would accept the request from the user, for example, to perform the viewer process. The actual request is handled or processed by the engine **object** component which generally resides on the server. The engine **object** component contains the horse power, or the processing power to process the request.

Detailed Description Text - DETX (159):
The engine **object** layer is generally located in the same or substantially same location as where the core technology or engine itself is being stored. Alternatively, the engine **object** layer and the engine may be optionally located in a distributed environment on different machines, servers, and the like.

Detailed Description Text - DETX (160):
FIG. 25 is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments. In FIG. 25, client 170 includes **object** manager **layer 172 with viewer process** 192. Client 170 executes the core technology 180, via **accessing engine object layer** 178 managed/stored on server 176, and communicated via server 174. Viewer process 190 is also optionally available to either or both servers 174, 176.

Detailed Description Text - DETX (161):
Client 182, located on the same server 176 as core technology 180 and engine **object** layer 178, may also be used to execute the core technology 180 and/or viewer process 192 via **object** manager layer 184. In this instance, the client 182 with the **object** manager layer 184 is located on the same server 176 as the engine **object** layer 178. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine **object** component layer, viewer process 194 and the **object** manager layer on the same machine 186, as well as the core technology.

Detailed Description Text - DETX (162):
Further, since the **object** manager is formatted or constructed of a client technology, the **object** manager can sit in a standard browser. This means that anyone that has an Internet browser, i.e., anyone that has access to the world wide web (WEB) can actually access the core engine technology and/or viewer process. Thus, by structuring the architecture of the present invention as described herein, users automatically become Internet, intranet and/or WEB enabled.

Detailed Description Text - DETX (163):
The present invention also transforms the core technology and/or viewer process from essentially client based technology into a client server and/or a thin client technology. This makes the core technology high level and/or viewer process accessible, thereby transforming any core technology and./or viewer process into client server, or hidden client technology. The browser is located on the client, and the browser leverages the **object** manager.

Accordingly, the browser optionally contains the **object** manager, and the **object** manager makes requests over, for example, the Internet, local network, and the like via a server, to the engine **object**. The server would be either a web server or a LAN server.

Detailed Description Text - DETX (164):

The present invention also advantageously provides the ability to have the client and the server, in a distributed environment as discussed above, or on the same machine locally. The present invention utilizes the DCOM communication protocol defining the communication protocol between the **object** manager and the engine **object** component. Accordingly, since DCOM can work on the same machine as well as in a distributed environment, DCOM does not necessitate that the engine **object or the object** manager component be on two separate machines.

Detailed Description Text - DETX (165):

FIG. 26 is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in network environments, such as the Internet. As illustrated in FIG. 21, **object** manager 14 communicates with engine **object** component 16, 18, 20 via DCOM specification and a networking environment, such as the Internet, intranet, and the like 168. In addition, **object** manager layer 14 also advantageously communications with viewer process 188a. Other types of component communication may also be utilized that provide the capability of a distributed component interaction over a networking environment.

Detailed Description Text - DETX (166):

FIG. 27 is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in the Internet environment. In FIG. 27, client 170 includes **object** manager layer 172. Browser/thin client 170a executes the core technology 180 and/or viewer process 192a, via accessing engine **object** layer 178 managed/stored on web server 176a, and communicated via the Internet 174a. Viewer process 190 is also optionally available to web server 176a.

Detailed Description Text - DETX (167):

Browser/thin client 182a, located on the same web server 176a as core technology 180, viewer process 192a and engine **object** layer 178, may also be used to execute the core technology 180 via **object** manager layer 184. In this instance, the browser/thin client 182a with the **object** manager layer 184 is located on the same web server 176a as the engine **object** layer 178. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine **object** component layer and the **object** manager layer on the same machine 186, as well as the core technology and viewer process.

Detailed Description Text - DETX (170):

Further, as indicated herein, the present invention may be used to automate and/or manually expedite the migration of a program specific Application

Programmer Interface from an original state into a generic interface by building an **object** for each engine. The **object** advantageously provides substantially uniform access to the engine and engine settings associated with the engine. The present invention may be applied across a broad range of programming languages that utilize similar concepts as described herein.

Detailed Description Paragraph Table - DETL (1):

class SomeEngineObject [//Wrapper Functions private: FARPROC_SomeFunction; BOOL SomeFunction.oval-hollow.; //EngineManagement protected: BOOL GetProcAddress (HINSTANCE, FARPROC&, LPCTSTR); BOOL GetProcAddresses.oval-hollow.; BOOL ProcessError.oval-hollow.; public: BOOL ActivateEngine (BOOL Activate); BOOL IsEngineActivated.oval-hollow.;];

Claims Text - CLTX (1):

1. A distributed computer implemented process for migrating at least one program specific Application Programmer Interface (API) from an original state into a substantially consistent interface by building an **object** for at least one of an engine and a viewer process, the **object** providing substantially uniform access to the at least one of the engine having engine settings and the viewer process, comprising the steps of:

Claims Text - CLTX (4):

(c) providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an **object** manager layer communicable with and managing the at least one engine component or the another viewer process via the substantially consistent interface.

Claims Text - CLTX (5):

2. A distributed computer implemented process according to claim 1, wherein the **object** manager layer is consistently communicable with each engine or the viewer process using the at least one engine component via the substantially consistent interface.

Claims Text - CLTX (6):

3. A distributed computer implemented process according to claim 1, wherein the substantially consistent interface comprises at least one of a component **object** module (COM) communication protocol and a distributed COM (DCOM) protocol.

Claims Text - CLTX (9):

6. A distributed computer implemented process according to claim 5, wherein the binary mode communication protocol includes a distributed component **object** module (DCOM) protocol.

Claims Text - CLTX (10):

7. A distributed computer implemented process according to claim 5, wherein

the binary mode communication protocol provides the capability for the engine component to execute independent of the **object** manager layer.

Claims Text - CLTX (12):
9. A distributed computer system for migrating at least one program specific Application Programmer Interface (API) from an original state into a substantially consistent interface by building an **object** for at least one of an engine and a viewer process, the **object** providing substantially uniform access to the at least one of the engine having engine settings and the viewer process, comprising:

Claims Text - CLTX (16):
a client configured to be connectable to said server and optionally configured to be connectable to another server, said client including an **object** manager layer communicable with and managing the at least one engine component or the another viewer process via the substantially consistent interface.

Claims Text - CLTX (17):
10. A distributed computer system according to claim 9, wherein the **object** manager layer is consistently communicable with each engine using the at least one engine component via the substantially consistent interface.

Claims Text - CLTX (18):
11. A distributed computer system according to claim 9, wherein the substantially consistent interface comprises at least one of a component **object** module (COM) protocol and a distributed COM (DCOM) protocol.

Claims Text - CLTX (21):
14. A distributed computer system according to claim 13, wherein the binary mode communication protocol includes a distributed component **object** module (DCOM) protocol.

Claims Text - CLTX (29):
17. A distributed computer implemented process including an **object** providing substantially uniform access to at least one engine having engine settings associated with the engine and an engine interface for accessing the engine settings and one or more features to be executed, comprising the steps of:

Claims Text - CLTX (31):
(b) providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an **object** manager layer communicable with and managing the at least one engine component via the substantially consistent interface.

Claims Text - CLTX (36):
a client configured to be connectable to said server and optionally configured to be connectable to another server, said client including an **object** manager layer communicable with and managing the at least one engine component stored on said server via the substantially consistent interface.

Claims Text - CLTX (39):
(b) mapping at least one engine function to at least one corresponding engine **object**;

Claims Text - CLTX (41):
(d) determining and matching arguments and returning values for mapping the at least one engine function to the at least one corresponding engine **object**; and

Claims Text - CLTX (43):
20. A computer implemented method utilizing a substantially consistent interface for individual **object** components that represent diverse technologies, comprising the steps of:

Claims Text - CLTX (45):
(b) at least one of substantially automatically and substantially uniformly, managing the individual **object** components using a predefined **object** manager and the consistent interface.

Claims Text - CLTX (50):
22. A distributed computer implemented process including an **object** providing substantially uniform access to at least one engine having engine settings associated with the engine and an engine interface for accessing the engine settings and one or more features to be executed, comprising the steps of:

Claims Text - CLTX (52):
(b) managing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an **object** manager layer to manage the at least one engine component via the substantially consistent interface.

US-PAT-NO: **6185555**

DOCUMENT-IDENTIFIER: US 6185555 B1

TITLE: Method and apparatus for data management using an event transition network

---------- KWIC ---------

US Patent No. - PN (1):
6185555

Application Filing Date - AD (1):
19981031

Drawing Description Text - DRTX (8):
FIG. 6 is a block diagram of the persistent **objects** comprising the system **objects** 122 (FIG. 1) in accordance with one possible relational embodiment of the present invention;

Drawing Description Text - DRTX (9):
FIGS. 7A and 7B are block diagrams of the persistent **objects** comprising the client **objects** 124 (FIG. 1) in accordance with one possible relational embodiment of the present invention;

Detailed Description Text - DETX (4):
The first tier is the data storage tier 110, which handles persistent storage of information through one or more databases 112 and 114. The second tier is the **object** tier 120, which captures and converts row data from the data storage tier 110 to **objects** and then executes business rules against the **objects**. The third tier is the user interface tier 140, which presents raw and/or **object** data to the system user via the display 104 and receives commands and data from the system user via the keyboard 106. These three tiers 110, 120 and 140 are "logical" because they are not confined to a single machine or process. Accordingly and as illustrated in FIG. 5, each of the tiers 110, 120 or 140 may simultaneously span many physical machines and processes across one or more networks.

Detailed Description Text - DETX (8):
The **object** tier 120 is divided into two distinct subsystems: system **objects** and processes 122, which control system operation and administrative functions; and client **objects** and processes 124, which control all data operations on the marketing data. The system **objects** 122 are "persistent **objects**," which are usually stored in the system database 112 and are instances of **classes** for

which a mapping is defined between the attributes of the **class** and columns in a database table. The values of the **object** attributes are stored in the database table through this mapping.

Detailed Description Text - DETX (9):

The **object** tier 120 may also include EOFLite components 126 to provide a "lightweight" binding layer for moving data back and forth between the client database 114 and minions 128 executing high-volume "batch processes." The EOFLite components 126 create instances of custom **classes** without having to live with the overhead inherent in the lower-level EOF frameworks, which are designed more for OLTP-type processing than for record-by-record batch processing.

Detailed Description Text - DETX (10):

Both the system **objects** 122 and the client **objects** 124 use persistent **object classes** to define specific business logic that is associated with the **object** attributes (and therefore implicitly with data from the databases 112 and 114 through the attribute-to-table-column mapping). Persistent **objects** can be instantiated from an arbitrary combination of database columns into "generic records," or can be instances of pre-defined **classes** containing attributes that are associated with particular columns in a table. Whether instances of generic records or instances of pre-defined **classes** are built is determined by the attribute-to-table mapping. In either case, the instances are created through a "database **access layer," and made available to custom processes** that can observe, manipulate, and save the instances back to the database.

Detailed Description Text - DETX (11):

The **object** tier 120 also contains minions 128, which provide services and sets of persistent **objects** to external client **objects**. In any given installation of the present invention, these **objects** can exist in both client-side end-user applications and in "interface-free" server-side applications called "agents." The agents exist to provide an executable wrapper around the minions 128. Thus, a particular agent can host an arbitrary combination of minions 128, and can be executed on any machine in a given installation of the present invention for which an executable has been built.

Detailed Description Text - DETX (12):

The minions 128 in these agent **processes use the database access layer** to convert database information between row form and persistent **object** form, and share these persistent **objects** with interface tier clients and/or other **object** tier clients. The minions 128 may also provide services that manipulate the **objects** locally and send the **objects** back to the database for storage. The minions 128 and persistent **objects** encapsulate business logic, and insulate the user applications 142 in the user interface tier 140 from data storage tier 110 dependencies. Minions 128 can exist in either remote agent processes or in the local address space of a client process. Client processes can also receive persistent **objects** directly from the database **access layer**.

Detailed Description Text - DETX (13):

There are two categories of minions 128 and persistent **objects in the object** tier 120. The first category consists of **objects** that oversee the control and administration functions of the present invention, such as the agent manager 130, job queue manager 132, notification manager 134, security guard 136, session management (not shown), and resource tracking and manipulation (not shown). The second category of **objects** imports, exports, reports on, cleans, maintains, and represents client data. All agent processes in the **object** tier 120 are controlled by a special minion called the "agent manager" 130. All access to remote agents and minions is granted through the agent manager 130.

Detailed Description Text - DETX (14):

The **object** tier 120 can be implemented using OpenStep **object** technology. This technology provides a database-independent framework for gathering information from local and/or remote database servers and converting it to **object** form. This technology also provides a relatively transparent **object** distribution mechanism that allows **objects** to communicate with each other across process and network boundaries as easily as if the **objects** coexisted in the same process. These key features directly support the ability for processes in the **object** tier 120 to execute on multiple physical machines.

Detailed Description Text - DETX (15):

The user interface tier 140 receives **object** data from the **object** tier 120, and controls the presentation of this data to the user. The user application 142 in the user interface tier 140 can operate in either two-tier or three-tier modes. User applications 142 that make direct connections to the database server to get data will be two-tier applications. User applications 142 that connect to agents in the **object** tier 120 to get data will be three-tier applications. It is possible for user applications 142 to directly include **object** tier **classes** and use them locally, operating in two-tier mode without actually bypassing the **object** tier 120. The user applications 142 may be implemented using native OpenStep applications, web-browser-based applications and applications built using technologies such as Microsoft Visual C++, which will communicate with the **object** tier via an **object** interface standard such as DCOM and CORBA.

Detailed Description Text - DETX (16):

Referring to FIG. 2, the basic anatomy of an agent process, in accordance with a preferred embodiment of the present invention, is designated generally by the numeral 150. The agent process 150 comprises a connection 152, an agent 154, an agent profile 156 and one or more minions 158, 160 and 162. As described above, the processes running in the **object** tier 120 are called agents 154, which are generic, independently executable programs that can run on any machine in the system. The sole purpose of an agent 154 is to serve as a distribution and control mechanism for an arbitrary combination of named **class** instances called minions 158, 160 and 162. The minions 158, 160 and 162 are **objects** that provide services and sets of persistent **objects** to external client **objects** through interfaces called minion formal protocols. There is a separate protocol defined for each possible minion **class** that needs to communicate across the network. Protocols are not **classes,** and are therefore never

instantiated or stored in a database.

Detailed Description Text - DETX (17):
Agents 154 do not inherently depend on the minions. During startup, an agent 154 extracts agent profile information 156 from a configuration file using a name given to it on the command line as a lookup key. The agent profile 156 specifies the minions 158, 160 and 162 that the agent 154 will manage, and any additional code that the agent 154 must load to support those **objects**. The agent process 150 configures itself dynamically based on the agent profile 156.

Detailed Description Text - DETX (18):
To support maximum flexibility on how systems can be configured, no restrictions are placed on where an agent 154 can execute (in other words, an agent 154 can be launched on any machine for which an agent executable has been built). This permits the logical **object** tier 120 (FIG. 1) to span the network across various physical machines concurrently. In practice, if agent 154 and/or the minions 158, 160 and 162 managed by the agent 154 depend on resources on a particular machine, the agent 154 should be configured to execute on that machine to keep those resources "local." External clients may gain access to the agent 154 via the connection 152, which is registered under a unique name given on the command line when the agent process 150 is launched.

Detailed Description Text - DETX (24):
The minion interfaces are defined in this way to make local minion instances 158, 160 and 162 and local minion proxy instances 188, 190 and 192 have the same apparent interface. The proxy **classes** provide a convenient place to perform client-side caching, a level of indirection through which to handle disconnection/reconnection with remote minions, and a **class** on the client side that can be associated with formal protocols. It also makes remote minions relatively transparent to developers. Client processes 182 that choose to use minions 158, 160 and 162 directly in their local address space operate in "two-tier" mode. Client processes 182 that choose to use minion proxies 188, 190 and 192 in their local address space operate in "three-tier" mode. In either event, the client processes 182 will be able to send the same messages to either type of **object**.

Detailed Description Text - DETX (25):
"Formal Protocols" define formal interfaces that are independent of any particular **class**. A protocol has a unique name, and declares a set of messages that can be implemented by any **class**. A particular **class** "conforms" to one or more protocols if it guarantees that it implements every message declared in each of the protocols. The interface to the minion **classes** is defined entirely in terms of formal protocols. A given minion **class** and its corresponding minion proxy **class** will both conform to the same formal protocol. A unique formal protocol is defined for every minion subclass.

Detailed Description Text - DETX (26):

Minion **classes,** minion proxy **classes,** and formal protocols are uniquely named to coexist in the same address space without interfering with each other. Client processes 182 can then simultaneously perform some operations in "two-tier" mode and others in "three-tier" mode depending on whichever is more appropriate.

Detailed Description Text - DETX (28):

Physical machine 202, having a keyboard 204, a monitor 206, a user interface tier 210 and a user application 212, is connected to a local area network 208. Physical machine 214, having a user interface tier 216, **object** tier 220 and data storage tier 222, is also connected to the network 208. The user application 212 of physical machine 202 is shown communicating with the agent manager 224 in the **object** tier 220 of physical machine 214. The agent manager 224 in turn communicates with agents 226, which communicate with database processes 228 and databases 230 in data storage tier 222.

Detailed Description Text - DETX (29):

Physical machine 232, having an **object** tier 234 and data storage tier 236, is also connected to the network 208. The user application 212 of physical machine 202 is shown communicating with the agent manager 238 in the **object** tier 234 of physical machine 232. The agent manager 238 in turn communicates with agents 240, which communicate with database processes 242 and databases 244 in data storage tier 236. The agents 240 of physical machine 232 are also shown communicating with the data storage tier 248, database processes 250 and databases 252 of physical machine 246 via the local area network 208.

Detailed Description Text - DETX (30):

Physical machine 254, having an **object** tier 256, is also connected to the network 208. The agent manager 258 and agents 260 of physical machine 254 are shown communicating with the database processes 242 of physical machine 232 and the database processes 250 of physical machine 246 via the local area network 208.

Detailed Description Text - DETX (36):

For example, each minion has a name that is used by a client process to request a connection to a specific minion through the agent manager 130. When the agent manager 130 is asked for connection information to a named minion, it performs the following steps: (1) consults the cross-reference to determine the generic name of the agent(s) that would be hosting that **object**; (2) examines all idle running agents with that generic name (if any), and determines if any of them report that their minion of the requested name is idle; (3) if any such agents are found, their connection information is loaded into an agent ticket and returned to the client; (4) if no agents with that generic name are running, the agent manager will simply start one and use it in the returned agent ticket; (5) if there are running agents with that generic name but none of them are idle, the agent manager will consult the present invention's system configuration file to determine how many agents with that generic name can be running simultaneously, and start a new one if limits permit; and (6) if all of that fails, no connection information is returned to the client (the client

must be prepared to handle a connection refusal).

Detailed Description Text - DETX (37):
Client processes that need to connect to a particular agent (or to some named minion running in a particular agent) will go through the agent manager 130 to get the information necessary to make the connection. This connection information is vended in the form of "agent tickets" (for connection to the agent itself) and "**object** tickets" (for connection to a minion), which can be "redeemed" by client **objects** to gain direct connections to remote agents and minions. Client processes must always get their connection information in "ticket" form through the agent manager 130. This is because the agent manager 130 will usually be controlling more than one instance of an agent with a particular generic name, each of which will be registered with the **object** distribution mechanism under a unique name that is only known to the agent manager 130 and the agent itself. The single exception to this is for connections to the AgentManagerAgent, which will always register under the name "AgentManagerAgent".

Detailed Description Text - DETX (43):
The NotificationCenterMinion serves as a central hub for routing advisory notifications from **objects** that "post" the notifications, to an arbitrary number of **objects** that register as "observers" of the notifications. Each advisory message has a name, and is associated with an **object** and a set of supplementary "user information." The names of the notifications are determined by the posting **objects,** and have to be known to observers so they can register as observers of the notifications. The **object** and "user information" associated with the notifications are also determined by the posting **object,** and can be whatever is deemed important for any particular advisory notification. Only one notification center will run in a given system of the present invention.

Detailed Description Text - DETX (45):
The security guard 136 hosts the SecurityGuardMinion, which is the first line of defense in the **object** tier for protecting the system against unauthorized access. The SecurityGuardMinion determines the validity of a given user/ID and password combination, and determines if a user has permission to perform a specific action. All guarded operations will obtain permission from the security guard before proceeding. The security guard 136 will always run in a stand-alone process so that it can be given special privileges if necessary. The security guard 136 thus prevents users from: (1) intentionally and/or unintentionally manipulating data without proper authorization; (2) reading data without proper authorization; and (3) making invalid changes to data whether authorized or not.

Detailed Description Text - DETX (46):
Due to the importance of the data contained within the databases and the efforts expended to obtain that data, the database must be protected against unauthorized access, yet still allow the present invention to perform tasks on behalf of those same users. Security becomes more difficult in the multiple

machine, multiple database architecture of the present invention because a user can connect to a database in a variety of ways: (1) directly from a present invention front-end process; (2) indirectly from an **object**-tier process to the present invention; (3) directly on the server via an SQL tool such as Informix DBaccess or Oracle SQLplus; and (4) directly from an arbitrary front-end development tool (such as VisualFoxPro) via ODBC.

Detailed Description Text - DETX (54):

The RoboDBAAgent hosts the RoboDBAMinion, which controls manipulation of database **objects**. The RoboDBAMinion provides services for: creating and dropping tables; creating and dropping indexes; unloading table schemas; and loading and unloading table data. The RoboDBAMinion can be embedded in an agent process and scheduled for deferred execution by the job management system. The RoboDBAMinion can also be given special database privileges that are not provided to any other process because the minion can run in a separate agent process. The only kinds of database tables that will be created through this minion are temporary database tables called "project tables" that provide work areas while processing the data for particular projects. Client production tables will always be created by external scripts when setting up the present invention for new clients.

Detailed Description Text - DETX (63):

The EventLogMinon provides a central point of access for all advisory messages that need to be stored persistently in the database. The "events" in the log are descriptions of some event in time, such as the observed failure of a critical component, or the failure of a job stream to complete. The events stored in the log describe unusual conditions that require the attention of a user or system operator. Initial deployments of the present invention will include an event log **object** as a simple local convenience **object** in each application that needs to access the log, rather than have a single central event log that runs in a remote agent.

Detailed Description Text - DETX (65):

Now referring to FIG. 6, a block diagram of the persistent **objects** comprising the system **objects** 122 (FIG. 1), in accordance with one possible relational embodiment of the present invention, is illustrated. The present invention is not limited to the specific system **objects** 122 described or to the relational embodiment illustrated in FIG. 6. Those persons skilled in the art will recognize that the present invention is designed to allow system **objects** 122 to be added, deleted or changed, and the relationships between the system **objects** 122 to be changed according to the intended application of the present invention. The relational nature of the system **objects** 122 allow the present invention to be modified without extensive recoding.

Detailed Description Text - DETX (66):

The system **objects** 122 comprise a number of interrelated persistent **objects that are the object** tier representation of rows in database tables. The persistent **objects** are grouped according to the minions that are primarily responsible for manipulating them.

Detailed Description Text - DETX (67):
Billing Persistent **Objects**:

Detailed Description Text - DETX (68):
Activity **objects** 302 associate an accounting activity code with a description. In this regard these **objects** are very similar to the CodeMaster 310 and CodeLook 312 is **objects**. Activity **objects** 302 are a separate **class** because they are closely associated with the accounting department, and are updated periodically with data obtained from the accounting system. The CodeMaster 310 and CodeLook 312 **objects** are relatively static and are associated exclusively with the present invention.

Detailed Description Text - DETX (69):
Billing **objects** 304 contain transaction information that is used to bill clients for services provided on their behalf. Billing **objects** 304 include such information as: an account number to bill against; tracking information about when the transaction occurred and when it was posted to the accounting department; a description of the services being billed; and billing units and amounts. Clients are billed for both processing storage and disk space usage. Billing **objects** 304 for processing are created at the end of process execution, whereas Billing **objects** 304 for disk storage are created on weekly intervals by the DiskUsageBillingAgent. A given Billing **object** 304 is associated with one MatterNum 308 and ClientBase 314 **object**.

Detailed Description Text - DETX (70):
BillingSummary **objects** 306 represent Billing **objects** 304 summarized over some period of time by client, project, matter number, and accounting activity code. These **objects** are created periodically by the BillingManagerMinion. Summarizing Billing **objects** 304 into BillingSummary **objects** 306 provides the accounting department with the minimum set of items needed to perform the required billing functions.

Detailed Description Text - DETX (71):
MatterNum **objects** 308 contain billing information and activity status for a particular ClientBase **object** 314. Typically, a ClientBase **object** 314 has a matter number for disk storage billing and a matter number for data processing billing. Project **objects** 316 also have a matter number for disk storage billing and a matter number for data processing billing. When a new Project **object** 316 is created for a given ClientBase **object** 314, the matter numbers for the single ClientBase **object** 314 are copied and associated with the new Project **object** 316. The matter numbers for the Project **object** 316 may be changed at any time after the Project **object** 316 is first created. Billing records are associated with one matter number.

Detailed Description Text - DETX (72):
Job Management Persistent **Objects**:

Detailed Description Text - DETX (73):
A JobConfig **object** 318 is a description of a process to be executed that may include: a text description of the job configuration; an identifier associating it with the process configuration **object** that was used to create the job configuration; the name of the agent to run; command line arguments to send to the agent when it is launched; the messages the agent must process; a description of any dependencies on other job configuration **objects**; tracking information about when the job configuration started and ended processing; and result information about the volume of records processed and success or failure of processing. A JobConfig **object** 318 is associated with one JobStream **object** 322 and one ProcessConfig **object** 324.

Detailed Description Text - DETX (74):
A JobQueue **object** 320 associates a single JobStream **object** 322 with a date, time, and priority at which to run, and context information to use during execution. Once a JobQueue **object** 320 has been saved in the database, it can be modified or deleted. Deleting the JobQueue 320 entry removes the associated JobStream 322 and JobConfig 318 **objects** from the database. Modifications that can be made to the JobQueue **object** 320 include changing the date, time or priority.

Detailed Description Text - DETX (75):
A JobStream **object** 322 associates a group of JobConfig **objects** 318 together into a stream, where the JobConfig **objects** 318 can have dependencies between them. A JobStream **object** 322 is associated with one or more JobConfig **objects** 318 and may include information such as a text description of the job stream, information about dependencies on other job streams, tracking information about when the job stream started and ended execution, control information about expected execution times, and job stream completion status information.

Detailed Description Text - DETX (76):
A ProcessConfig **object** 324 is a "template" from which JobConfig **objects** 318 are created. When a system user of the present invention wants to submit a new JobConfig **object** 318 to the system, an existing ProcessConfig **object** 324 can be used as a template, or a new ProcessConfig **object** 324 can be created. In either case the ProcessConfig **object** 324 is saved back to the database. ProcessConfig **objects** 324 are essentially JobConfig **objects** 318 without the dynamic attributes, but may include such information as a text description of the process configuration, the name of the agent to run, command line arguments to send to the agent when it is launched, the messages the agent must process, and a description of any dependencies on other job configuration **objects**. A ProcessConfig **object** 324 is associated with one ProcessStream **object** 326. A ProcessStream **object** 326 and its associated ProcessConfig **objects** 324 never become part of the job queue. Instead, when a user submits a JobStream **object** 322 to the job queue for execution, the ProcessStream 326 and ProcessConfig 324 **objects** are used to create JobStream 322 and JobConfig 318 **objects; those are the objects** actually submitted to the queue.

Detailed Description Text - DETX (77):
A ProcessStream **object** 326 is a "template" from which JobStream **objects** 322 are created. When a system user wants to submit a new JobStream **object** 322 to the system, an existing ProcessStream **object** 326 can be used as a template, or a new ProcessStream **object** 326 can be created. In either case, the ProcessStream **object** 326 is saved back to the database. ProcessStream **objects** 326 are essentially JobStream **objects** 322 without the dynamic attributes, but may include such information as a text description of the job stream, and information about dependencies on other job streams.

Detailed Description Text - DETX (78):
Media Processing And Tracking Persistent **Objects**:

Detailed Description Text - DETX (79):
A CartridgeTape **object** (not shown) differs from basic Tapes in that it has compression and track count information associated with it. A given CartridgeTape **object** can be associated with one ClientBase **object** 314 (but a given ClientBase **object** 314 may be associated with many CartridgeTape **objects,** or more generally, with many other PhysicalMedia **objects** 332).

Detailed Description Text - DETX (80):
DataFile **objects** 328 represent tracked data files in the system and may include files imported from external media, files created as a side-effect of a system or client process, files extracted from a database, and files created outside the system (such as a word processor document) that have been entered into the system by a user for tracking purposes. The DataFile **object** 328 records sundry information of interest about the file. A given DataFile **object** 328 is associated with one Project **object** 316. The lifetime of a DataFile **object** 328 is specified by the Project **object** 316 it is associated with.

Detailed Description Text - DETX (81):
DataTable **objects** 330 represent tracked database tables in the system. The database tier contains both "production" tables (i.e. tables whose formats are fixed for a given client), and "project" tables that are created as a side-effect of processing the client data. The DataTable **object** 330 records sundry information of interest about the table. A given DataTable **object** 330 is associated with one Project **object** 316. The lifetime of a DataTable **object** 330 is specified by the Project **object** 316 it is associated with.

Detailed Description Text - DETX (82):
NineTrackTape **objects** (not shown) represent single-reel, nine-track tapes, that are usually associated with mainframe computers. NineTrackTape **objects** differ from basic Tapes in that they have tape density information associated with them. A given NineTrackTape **object** can be associated with one ClientBase **object** 314 (but a given ClientBase **object** 314 may be associated with many NineTrackTape **objects,** or more generally, with many other PhysicalMedia **objects**

332).

Detailed Description Text - DETX (83):
A PhysicalMedia **object** 332 describes the common attributes of physical media used for transporting data (such as magnetic tapes). These common attributes include not only characteristics of the physical media itself (such as its volume label), but also sundry information such as where it is physically located, when it was received, general comments, etc. A particular ClientBase **object** 314 can be associated with many PhysicalMedia **objects** 332 (but each of those can only be associated with a single ClientBase **object** 314). All PhysicalMedia **objects** 332 are cataloged by the Librarian so they can be tracked. PhysicalMedia **objects** 332 almost always have a finite lifetime, after which they are disposed of in some manner that is determined at the time they are cataloged. The lifetime of a PhysicalMedia **object** 332 is determined by the Project **object** 316 it is associated with.

Detailed Description Text - DETX (84):
A Tape **object** (not shown) represents any kind of magnetic tape media such as CartridgeTapes and NineTrackTapes. A Tape **object** can be thought of as an abstraction that collects common attributes shared by all tape **objects**. A TapeDrive **object** 334 represents tape drives in which magnetic tapes can be read and written. A TapeRack **object** 336 represents the racks in which magnetic tapes are stored. A TapeSlot **object** 338 represents a slot in a tape rack.

Detailed Description Text - DETX (85):
Miscellaneous Persistent **Objects**:

Detailed Description Text - DETX (86):
An Event **object** 340 represents an event that happened on the system that needs to be stored persistently in the event log database table. Event **objects** 340 describe what happened, when it happened, what operation or process was being performed at the time the event occurred, and some notion of severity (although severity is hard to convey because it is usually a function of who is observing the event). Because Event **objects** 340 are associated with processes, they are implicitly associated with a particular ClientBase **object** 314 unless the Context under which the process was operating did not include a ClientBase **object** 314 (which is the case for processes operating on behalf of the present invention itself). UserEventAssoc **objects** 374 contain the Event **objects** 340 that each User **object** 360 is interested in.

Detailed Description Text - DETX (87):
A CodeLook **object** 312 provides a long description for a given "code" (i.e. a short single word description or mnemonic), and associates the code with a CodeMaster **object** 310. All codes used in the system are stored in a single CodeLook table in the database. The codes that are grouped together are associated with the same CodeMaster **object** 310.

Detailed Description Text - DETX (88):
A CodeMaster **object** 310 describes the different codes that are used by the system. This description includes the name of the table column associated with the code, a long description for the code, the length of the code, and whether or not new codes can be added to the list of codes associated with this code master. The possible code values for a given CodeMaster **object** 310 are stored in the CodeLook table, and are associated with the CodeMaster **object** 310 through its unique identifier.

Detailed Description Text - DETX (89):
DMAMail **objects** 342 represent customers who do not want to receive promotion pieces from a mail marketing campaign. DMAPhone **objects** 344 represent customers who do not want to be contacted by phone on behalf of a telemarketing campaign.

Detailed Description Text - DETX (90):
A NotifyRegistry **object** 346 associates a notification name with a user who wants to have an e-mail and/or pager message sent to them when the notification occurs. NotifyRegistry **objects** 346 may include the name of the notification to dispatch, the user to dispatch the message to, and the transport mechanism to use to send the message (i.e. e-mail or pager).

Detailed Description Text - DETX (91):
A MinionLog **object** 348 represents a log message written to the MinionLog table by a minion. The MinionLog table is a general purpose area where any minion can store persistent advisory messages. A MinionLog **object** 348 may include the name of the minion that posted the message, when the activity occurred, an "activity code," an arbitrary message, and the ClientBase **object** 314 and/or Project **object** 316 that the minion was operating on when the message was saved

Detailed Description Text - DETX (92):
A State **object** 350 simply associates a standard abbreviation for a state that is a member of the USA with the full state name. A ZipCity **object** 352 is a utility **object** that associates zip codes, counties, and states together.

Detailed Description Text - DETX (93):
Security Persistent **Objects**:

Detailed Description Text - DETX (94):
An Action **object** 354 describes the operations that a user can perform with the system. These Action **objects** 354 can be grouped into ActionGroup **objects** 356. A given Action **object** 354 can be a member of zero or more ActionGroup **objects** 356. A given UserID is associated with a ClientBase **object** 314 and an Action **object** 354 through a Permission **object** 358. In other words, a particular user can only perform a certain Action **object** 354 on a ClientBase **object** 314 if there is a Permission **object** 358 for it.

Detailed Description Text - DETX (95):
An ActionGroup **object** 356 groups one or more actions together. ActionGroup **objects** 356 are an administrative convenience for assigning a collection of Action **objects** 354 to a UserID and ClientBase **object** 314 rather than having to assign them one at a time.

Detailed Description Text - DETX (96):
A Permission **object** 358 associates a User **object** 360 with a ClientBase **object** 314 and an Action **object** 354. There will be one of these **objects** for every Action **object** 354 that a particular User **object** 360 is allowed to perform on a given ClientBase **object** 314.

Detailed Description Text - DETX (97):
Session Management Persistent **Objects**:

Detailed Description Text - DETX (98):
A ClientBase **object** 314 contains database and system resource management information for the different clients. This information may include a client name, database connection and access information, database status, default billing information, directory locations for data files associated with the client base **object**, and default resource expiration information. ClientBase **objects** 314 may have zero or more Project **objects** 316 associated with them. A particular ClientBase **object** 314 and its associated Project **object** 316 are only accessible by User **objects** 360 that have permission to work with those Project **objects** 316. A ClientBase **object** 314 may have a finite lifetime (i.e. it may only remain active for a period of time, then be taken off the system), or may be essentially perpetual.

Detailed Description Text - DETX (99):
DataSpace **objects** 362 describe the data spaces on disk where the database servers store table and index data. DataSpace **objects** 362 may describe the name of the data space, whether the data space is a "project" table data space or a "production" table data space, a brief description of the data space, and the ClientBase **object** 314 associated with the DataSpace **object** 362.

Detailed Description Text - DETX (100):
A Project **object** 316 is a named collection of work associated with a ClientBase **object** 314. A Project **object** 316 may include a description of the project, a project code and status, archival information, and information about where to store flat files associated with the project. Project **objects** 316 can be associated with one ClientBase **object** 314, zero or more DataTable **objects** 330, zero or more DataFile **objects** 328, zero or more PhysicalMedia **objects** 332, zero or more JobStream **objects** 322 and ProcessStream **objects** 326, zero or more Session **objects** 364, one MatterNum **object** 308 for disk usage billing, and one MatterNum **object** 308 for processing billing.

Detailed Description Text - DETX (101):
All User **objects** 360 that have access to a particular ClientBase **object** 314 are able to access all of its Project **objects** 316. Multiple Sessions (and therefore multiple User **objects** 360) may work on a Project **object** 316 concurrently.

Detailed Description Text - DETX (102):
A Session **object** 364 is a combination of a particular Computer and Context that represents a connection from a particular workstation to a project on behalf of a User **object** 360. A Session **object** 364 is always associated with exactly one Project **object** 316 and therefore with exactly one ClientBase **object** 314. Sessions are created when User **objects** 360 connect to the present invention from some workstation application, and are used in nearly all other system messaging to establish the context in which those messages should execute. A User **object** 360 on a particular workstation computer may establish exactly one Session from that workstation. However, a User **object** 360 may establish concurrent sessions from different workstations.

Detailed Description Text - DETX (103):
A SysDefault **object** 366 stores default values for the present invention. These default values consist of key-value pairs in the form of strings. A UserDefault **object** 368 stores user-dependent default values for the present invention. These default values consist of key-value pairs in the form of strings, and can vary from one user to the next.

Detailed Description Text - DETX (104):
A User **object** 360 represents a user of the present invention. A User **object** 360 is allowed to work on selected ClientBase **objects** 314, and has access to all Project **objects** 316 associated with those ClientBase **objects** 314. A user may connect to (i.e. establish a Session with) the present invention from any workstation computer that is able to communicate with the system, and may have concurrent Sessions from multiple workstations, limited to a single Session on a particular workstation.

Detailed Description Text - DETX (105):
The Expression **object** 370 contains expression and function information used by applications. The ClientExprAssoc **object** 372 contains each valid expression for each ClientBase **object** 314. The eo_sequence_table 376 keeps track of the last original identifier assigned to each **object**. The SecurityGuard table 378 contains the userid and encrypted password for each valid user of the system. Only the security guard 136 (FIG. 1) has access to this table.

Detailed Description Text - DETX (106):
Client **Objects**

Detailed Description Text - DETX (107):

Referring now to FIG. 7A, a block diagram of the persistent **objects** comprising the client **objects** 124 (FIG. 1), in accordance with one possible relational embodiment of the present invention, is illustrated. The present invention is not limited to the specific client **objects** 124 described or to the relational embodiment illustrated in FIG. 7A. Those persons skilled in the art will recognize that the present invention is designed to allow client **objects** 124 to be added, deleted or changed, and the relationships between the client **objects** 124 to be changed according to the intended application of the present invention. The relational nature of the client **objects** 124 allow the present invention to be modified without extensive recoding.

Detailed Description Text - DETX (199):

Now referring to FIG. 7B, another block diagram of the persistent **objects** comprising the client **objects** 124 (FIG. 1), in accordance with one possible relational embodiment of the present invention, is illustrated. The following tables are similar in that none of them are used to store client data. Instead, they are used to help direct the application software processes that need to validate and load incoming client data.

Detailed Description Text - DETX (203):

Eo_sequence_table 490 assigns the next sequential value to a given table's primary key column. In the present invention, every table has an "artificial" primary key called an "**Object** ID" (OID). The value stored in the primary key of each table is a system generated sequential number. The application uses eo_sequence_table 490 to determine the next primary key value that is to be assigned for each table in the client **objects** 124.

Detailed Description Text - DETX (204):

DDColumn 492 is one of three tables used together to group related column names. DDColumn 492 stores the name of each column used throughout the client **objects** 124. Along with the column name, it also stores a brief description and what "type" of data it contains (e.g. DATE, CHAR(8), INTEGER, etc. . .).

Detailed Description Text - DETX (211):

Referring to FIG. 8, a block diagram of the EOFLite components 126 (FIG. 1), in accordance with a preferred embodiment of the present invention, is illustrated. The EOFLite components 126 are designed to provide a "lightweight" binding layer for moving data back and forth between the client database 114 and high-volume "batch" processes. The EOFLite components 126 create instances of custom **classes** without having to live with the overhead inherent in the lower-level EOF frameworks, which are designed more for OLTP-type processing than for record-by-record batch processing.

Detailed Description Text - DETX (212):

The EOFLite components 126 are designed to work with the same enterprise **object classes** that are used with EOF, so that a single **object** model can be developed that works the same for both interactive and batch processing (i.e. the same business logic will be used by both). The major **classes** in EOFLite

are designed to model familiar database concepts, such as the database itself, connections to the database, cursors in a connection, and bindings between the data being manipulated by a cursor and some **object** that is to be populated with that data. These components supplement the basic EOF "primitive" **classes,** such as EOModel 602, EOQualifier 604, EOEntity 606, and EOAttribute 608, and when used together result in a much more efficient mechanism for performing batch processing. The EOFLite components 126 are as follows:

Detailed Description Text - DETX (216):
EOAttribute 608 represents the table column whose data the receiver is binding to its **object**.

Detailed Description Text - DETX (217):
DBDatabase 610 represents a single physical database. Each instance of this **class** can be associated with zero or more DBConnection 612 instances.

Detailed Description Text - DETX (219):
DBCursor 614 represents data cursors operating through a connection. These instances are configured for a single entity and an associated **class of objects** whose instances will be populated with the data retrieved by the cursor.

Detailed Description Text - DETX (220):
DBBinding 616 represents bindings between a single EOAttribute 608, which represents a single column in a table, and a single instance variable of an **object**. These "bindings" are used by instances of the DBCursor 614 to map data between the attributes of an EOEntity 606 and the instance variables of the cursor's prototype **object**.

Detailed Description Text - DETX (221):
NSObject 618 is the **object** to and from which data is being mapped by the receiver.

Detailed Description Text - DETX (222):
String is a lightweight implementation of NSString that provides more external access to the internal representation of the string, and minimizes the amount of memory allocation overhead that takes place during string manipulation. This **class** is primarily provided as an optimization for batch processes that cannot afford the performance overhead of the NSString **class**.

Detailed Description Text - DETX (223):
Number is a lightweight implementation of NSNumber that minimizes the amount of memory allocation overhead that takes place during data manipulation. This **class** is primarily provided as an optimization for batch processes that cannot afford the performance overhead of the NSNumber **class**.

Detailed Description Text - DETX (224):
CalendarDate is a lightweight implementation of NSCalendarDate that minimizes the amount of computation overhead involved to manipulate dates. This **class** is primarily provided as an optimization for batch processes that cannot afford the performance overhead of the NSCalendarDate **class**.

Detailed Description Text - DETX (266):
An agent description consists of an agent name with an associated agent profile (discussed in a previous section), list of **object** descriptions, and list of additional frameworks to load to support the **objects** the agent will manage.

Detailed Description Text - DETX (267):
The System Configuration File is used both by interface-layer minion proxies (to locate their corresponding remote **objects), and by object**-layer agents (to determine their profile). This means that the System Configuration File must be in at least one shared location that is accessible across all machines that participate in the system.

Detailed Description Text - DETX (272):
Client-specific executable files for context-dependent agents are located in a path that is obtained from the client **object** in the context of the session that is making use of the agent (specifics forthcoming).

US-PAT-NO: **5960200**

DOCUMENT-IDENTIFIER: US 5960200 A

TITLE: System to transition an enterprise to a distributed infrastructure

---------- KWIC ---------

US Patent No. - PN (1):
5960200

Application Filing Date - AD (1):
19960916

Brief Summary Text - BSTX (9):
A preferred embodiment of the invention includes a system to transition an entire business enterprise to a distributed infrastructure. The distributed infrastructure is preferably a multi-tiered client/server target architecture that adheres to open system standards. The multi-tiered architecture preferably includes at least four **layers including a separate process control layer** and functionality layer. The **process control layer** includes a state router to control work flow in accordance with the business procedures of the enterprise. The functionality layer includes modules for performing the work. The architecture also preferably includes a presentation layer for interfacing with a user, and a data retrieval **layer for accessing** data stored in a separate data storage layer.

Drawing Description Text - DRTX (2):
The foregoing and other **objects,** features and advantages of the invention, including various novel details of construction and combination of parts will be apparent from the following more particular drawings and description of preferred embodiments of a system to transition an enterprise to a distributed infrastructure in which the reference characters refer to the same parts throughout the different views. It will be understood that the particular apparatus and methods embodying the invention are shown by way of illustration only and not as a limitation of the invention, emphasis instead being placed upon illustrating the principles of the invention. The principles and features of this invention may be employed in various and numerous embodiments without departing from the scope of the invention.

Drawing Description Text - DRTX (9):
FIG. 7 is a schematic diagram illustrating the business **process layer** 120, functionality **layer 130, and data access layer** 140 of FIG. 1.

Drawing Description Text - DRTX (32):
FIG. 30 is a block diagram of the graphical business **object** editor 330 of FIG. 1.

Detailed Description Text - DETX (3):
A preferred multi-tiered architecture 10 of the present invention comprises at least four separate layers. As illustrated, the architecture 10 includes at least one presentation **layer 110, one separate business process layer** 120, one separate functionality **layer 130, one separate data access layer** 140, one separate data storage layer 150, and one separate control layer 160, all inter-connected through communication links 170. The data storage layer 150 preferably includes a user interface repository 152, a business process repository 154, a business **object** repository 156, and a data record repository 158 for storing data.

Detailed Description Text - DETX (5):
A preferred re-engineering system includes a graphical user interface editor 310, a graphical business process editor 320, a graphical business **object** editor 330, a graphical data editor 340, a logic development environment 350, and facilitation tools 360. The logic development environment 350 is in communication with the functionality layer 130 of the multi-tier architecture 10. The graphical user interface editor 310, the graphical business process editor 320, the graphical business **object** editor 330, and the graphical data record editor 340 are in communication with the user interface repository 152, the business process repository 154, the business **object** repository 156, and the data record repository 158, respectively.

Detailed Description Text - DETX (8):
FIG. 2 is a functional block diagram of the interrelationships of FIG. 1. Conceptually, the user interface translator 210 and the graphic user interface editor 310 affect the presentation layer 110 of the multi-tiered architecture 10. The graphical business process editor 320 affects the business **process layer** 120. The procedural language conversion utility 220, the graphical business **object** editor 330, and a logic development environment 350 conceptually affect the functionality layer 130. The procedural language conversion utility 220 also conceptually affects the data **access layer** 140. The data definition language translator 230 and the graphical data record editor 340 conceptually affect the data storage layer 150.

Detailed Description Text - DETX (28):
A preferred embodiment of the present invention uses a frame-type representation in an **object**-oriented organization in which the frames represent **objects**. More specifically, the frames representing general concepts are referred to as **classes** and those representing specific occurrences of a concept are referred to as instances. In this context, attributes are termed members, and member inheritance and procedural attachment take place as in a frame system.

Detailed Description Text - DETX (29):
In **object**-oriented systems, however, **objects** communicate with one another by sending and receiving messages. When an **object** receives a message, it consults its pre-defined answers for messages to decide on what action to take. These answers can be stored directly with the **object** or inherited from a higher level **object** somewhere in the network hierarchy. Usually, the action involves triggering some rules, executing procedural code, or sending new messages to other **objects** in the system.

Detailed Description Text - DETX (30):
Similarly to the display platform user interface representation structures 118, the application user interface representation structures 116 store descriptive information representative of the different **objects** that compose a user interface. Each **object** is described by a structure comprising a plurality of fields containing information representing an attribute of that **object** or a relationship between the **object and another object**. The user interface engine 117 maps each of the different **objects** that compose the user interface of a given application into the corresponding representations 118 in the user interface display platform 115 of choice for that application.

Detailed Description Text - DETX (33):
FIG. 5 is a schematic diagram of a sample mapping between application user interface representation structures 116 and display platform user interface representation structures 118. In the figure, the user interface display platform 115 is exemplified as Microsoft Windows 3.x and the display platform user interface representation structures 117 are thus the internal Windows 3.x management structures. However, other user interface display platforms 115 using similar internal structures to manage windows are supported by the exact same user interface engine 117. Notably, the internet's world-wide web, based on the HTML or Java user interface languages, is another example of user interface display platform 115. Indeed, in a preferred embodiment of the present invention, the user interface engine 117 is written using Microsoft Visual C++ and based on the industry-standard Microsoft Foundations **Classes** (MFC) **class** library, which allows cross-platform development for Windows 3.x, Windows 95, Windows NT, MacOS, and UNIX-based user interface display platforms 115, including internet web servers.

Detailed Description Text - DETX (35):
During initialization, the user interface engine 117 first initializes its initial state, setting up any structures necessary for operation. Depending on the implementation, the user interface engine 117 can then initialize communications with the business process layer 120, receiving a client identification number. Depending on the implementation, the user interface engine 117 can also display an initial application menu or screen, initial **objects** that are provided by the business process layer 120.

Detailed Description Text - DETX (36):
After completing the initialization, the user interface engine 117 continues

to the user input module 117-2. The user interface engine 117 waits for user input and processes it accordingly. In particular, the user input module 117-2 handles interactions with GUI **objects** and performs application-dependent actions in response to user inputs.

Detailed Description Text - DETX (40):

Essentially, during idle message polling the user interface engine 117 queries the state router for any initial messages. At the start of an interactive application, a first screen needs to be displayed to the user. This screen is usually a sign-on, or logon, screen which contains fields for the user identifier and user password, with possibly peripheral buttons to change the user password and access help screens. In addition, other graphics, such as an application logo or wallpaper, might be decorating the screen. After these initial messages have been processed, resulting in the display of the logon screen, application menu, and other **object** for the user to act upon, the user interface engine 117 waits to process user inputs. If the user takes no action and idle message polling is enabled, the user interface engine 117 will periodically query the state router for any messages. If message polling is disabled, the user input loop will continue indefinitely. Using a window mapping structure, which is preferably a two-way associative array, it is possible for the user interface engine 117 to allow window control handlers of the user interface display platform 111 to manage general window operation and make callbacks to the user interface engine handlers when an action is required, for example, when a button is pressed.

Detailed Description Text - DETX (41):

In a preferred embodiment of the present invention, the user interface engine 117 can process any type of action from any type of screen **object,** e.g. a button being pressed, a control gaining the input focus, or the Tab key being pressed. Typically, when an action is performed, one of two things may happen: the user interface engine 117 performs some internal function based on the action, or sends information to be processed back to the business process layer 120. In a particular preferred embodiment of the invention, all actions are referred back to the business process layer 120 for processing, along with any updated field values.

Detailed Description Text - DETX (42):

FIG. 7 is a schematic diagram illustrating the business **process layer** 120, functionality **layer 130, and data access layer** 140 of FIG. 1. In a preferred embodiment of the invention, these layers can be hosted on similar platforms. In a preferred embodiment of the present invention, these platforms include a host processor 132, in which the various engines are resident, internal or external storage 134, on which the logic or data access server runtime environment resides, and a terminal console 136 which serves as a human interface for host administration purposes. In addition, a communications controller 138 such as a LAN controller, modem or similar device serves as an interface to a communication link. The host computer system 132 can be considered conventional in design and may, for example, take the form of a E55 workstation, manufactured by Hewlett Packard Corporation. As shown, the business **process layer** 120, functionality **layer 130, and data access layer** 140

further include a printer 135 which can be used to provide a permanent record of application log files, reports, source code, or process **objects** and flows according to the present invention.

Detailed Description Text - DETX (47):

The unpacking procedure converts the request string into an array of request application user interface representation structures 121. This array is then passed to a main state router 122-1 function, which accounts for the core processing of the state router 122. Routing logic 122-2 then directs the **objects** to servers in the functionality layer 130 or the database layer 140. Once the state router 122 completes its processing, the resulting array of return application user interface representation structures 121 is again packed into a return string, which is passed back to the user interface engine 117 using an RPC mechanism 122-9.

Detailed Description Text - DETX (48):

Because new application user interface representation structures 121 can be added to facilitate the transport of new types of **objects** as required by a particular application, the packing and unpacking functions include a library having primitives which pack and unpack bytes (8-bit integers), words (16bit integers), double words (32-bit integers), and strings (both variable- and fixed-length). To create a new application user interface representation structure 121, a developer need only create packing and unpacking routines for that structure, assembling these functions from the primitive routines.

Detailed Description Text - DETX (49):

In the preferred embodiment of the present invention, the packing and unpacking library is written in such a way that the same source code compiles using structures (under ANSI C) or using **object classes** (under ANSI C++). Although the ANSI C language interface is very usable, the ANSI C++ language interface makes use of **object**-oriented features such as virtual functions to make packing and unpacking as transparent as possible. High-level packing and unpacking routines take arrays (or, in ANSI C++, containers) of application user interface representation structures 121 and create a single character string containing the packed information suitable for RPC transmission. This string contains type information as well as member data, so that any sequence of application user interface representation structures 121 can be sent and properly reconstructed at the receiving end.

Detailed Description Text - DETX (65):

FIG. 11 is a block diagram of the functionality layer 130 of FIG. 1. Conceptually, the functionality layer 130 manages the business **objects** manipulated by the business process layer 120. These business **objects** constitute the fundamental components of an application. In a traditional manual system, a business **object** is associated with one or more physical paper forms. These forms contain the fields that hold the information relevant to the business **object**. Forms differ not only in their physical appearance, but also in the rules that govern their use. For example, a highly confidential form is treated differently from a non-confidential form. Other business rules

may also govern the handling of forms. For example, some invoices might require more than one signature if their amount is bigger than a certain value. It is the form and the rules that govern its handling that define a business **object** in a traditional manual system.

Detailed Description Text - DETX (66):

The business **objects** represent the physical forms, the information in those forms, and the rules that govern these forms. As discussed previously in the context of a re-engineering or custom-developed application, the business process engine 124 of the business process layer 120 manages the flow of business **objects,** interprets their rules, and acts on these rules.

Detailed Description Text - DETX (74):

FIG. 12 is a block diagram of the data access layer 140 of FIG. 1. The data access layer 140 comprises a number of data servers. As mentioned previously, the data servers are used to access and retrieve data from the data storage layer 150. Preferably, one data server exists for each of the four application **object** repositories. Consequently, there is user interface data server 141 to manipulate user interface **objects** 142, a business process data server 143 for business processes 144, a business **object** data server 145 for business **objects** 146, and a database server 147 for application data records 148. The data servers constitute the sole interface between the data storage layer 140 and the functionality layer 130, and each data server is only in charge of exchanging with the functionality layer 130 information about the type of application **object** it services.

Detailed Description Text - DETX (78):

The data access layer 140 can now be viewed as a set of database access and retrieval functions. There is one function for each one of the data access construct of the source Data Base Management System (DBMS). Each such function must emulate the behavior of the corresponding source DBMS data accessor construct. In the preferred embodiment of the present invention, the target DBMS is typically based on a conventional relational model, and is called a Relational DBMS (RDBMS). The source DBMS can be built around a number of conventional data models. The most common such models are the flat file, hierarchical, network, CODASYL (Conference on Data Systems Languages), relational, and **object**-oriented data models.

Detailed Description Text - DETX (84):

FIG. 13 is a block diagram of the data storage layer 150 of FIG. 1. The data storage **layer 150 is a repository for data accessed by the data access layer** 140. The user interface data repository 152 provides user interface **objects** 153 to the data **access layer** 140. The business process data repository 154 provides business **processes 155 to the data access layer** 140. The business **object** data repository 156 provides business **objects** 157 to the data **access layer** 140. The application data records repository 158 provides data records 159 to the data **access layer** 140.

Detailed Description Text - DETX (86):
A preferred embodiment of the present invention uses the relational data model for the data storage layer. The relational data model can be viewed at three different levels: conceptual, logical, and physical. The conceptual level consists of entities, attributes, and relations. Entities are things that exist on their own, distinguishable from other **objects**. Entities (records, rows) are described in terms of their attributes (fields, columns). Relations are common fields between entities used to connect entities together. The Entity-Relationship data model (ER) is the predominant conceptual level description tool. It is used as a diagramming technique where rectangles represent entities, circles represent attributes, diamonds represent relationships. The logical level consists of records, fields, and relations.

Detailed Description Text - DETX (98):
Configuration management 167 provides libraries for a number of diverse purposes. For instance, application version management functions are provided. In addition, currency is handled through locking functions to insure data consistency. Data integrity is controllable at the functionality layer by the business **objects** rules or constraints. The underlying database management system usually provides data integrity controls implicitly available to applications, but the usage of such controls are not recommended because it would mean encoding business logic in the data layer 150 instead of confining it to the functionality layer 130, and would therefore be contrary to the fundamental principle of the preferred multi-tiered architecture.

Detailed Description Text - DETX (137):
FIGS. 23A-23B are C and C++ target code fragements 227', 227", respectively, for the source code fragment of FIG. 18. As illustrated, the intermediate language illustrated in the output file 225' of FIG. 21 is transformed into a select target language source code 227. The target language can be any procedural programming language (such as C) or any **object**-oriented programming language (such as C++). As described, a different second phase transformer program 226 is used to generate source code in each target language.

Detailed Description Text - DETX (155):
As mentioned previously with regard to FIG. 1, the custom and re-engineering system 30 focuses on providing an enterprise a facility for maintaining and enhancing distributed infrastructure. Even though this facility is an integral part of the overall system of the present invention, it is really an add-on facility that becomes paramount once the transition is complete. Consequently, only an overview of the custom and re-engineering system 30 will be provided here. At a high-level therefore, the custom and re-engineering system 30 includes a graphical user interface editor 310, a graphical business process editor 320, a graphical business **object** editor 330, a graphical data record editor 340, a logic development environment 350, and facilitation tools 360.

Detailed Description Text - DETX (157):
FIG. 28 is a block diagram of the graphical user interface editor 310 of FIG. 1. The graphical user interface editor 310 is a typical user interface

made to create menus and paint screens. As such, the graphical user interface editor 310 includes a screen editor 311 to position graphical representations of business **objects** on a screen or form. Screens can thus include text fields, labels, buttons, selection boxes, pull down lists, and similar graphical **objects** that compose a user interface. These graphical representations of business **objects** can be grouped so that a screen can be composed of sub-screens. This is useful to represent screen overlays, which are screens that have a fixed area as well as a variable area that changes depending on user actions. Sub-screens are also useful for grouping together business **objects** that need to be displayed across a number of application screens. The screen editor 311 creates internal user interface representations 312 which are processed by a user interface code generator 313 into data stored in the user interface repository 152.

Detailed Description Text - DETX (162):

FIG. 30 is a block diagram of the graphical business **object** editor 330 of FIG. 1. The graphical business **object** editor 330 is a tool that enables the graphical editing of business **objects** and their relationships. A business **object** editor 331 generates internal business **object** representations which are converted by a business **object** code generator into data stored in the business **object** repository 156.

Detailed Description Text - DETX (163):

In a preferred embodiment of the present invention, the graphical business **object** editor 330 can be viewed as a Entity-Relationship (ER) diagramming tool. The ER data model is the predominant conceptual level description tool and is used as a diagramming technique where rectangles represent entities, circles represent attributes, diamonds represent relationships. This graphical ER diagram can be used to generated automatically the application database schema, and a number of the basic data accessor queries.

Detailed Description Text - DETX (164):

In addition, default constraints can be automatically associated to business **objects** based on the business **object** type. This can lead to automatic generation of maintenance screens for lookup business **objects** that can take a known range of values. The graphical business **object** editor can also be used to create templates that can be reused throughout an application. For instance, every screen may have a number of fixed function keys or buttons such as display, insert, delete, update, clear, refresh, backup, or quit, as well as a number of variable function keys whose semantics change from screen to screen. These function keys can be treated as a group and provided automatically as part of the template for every screen in an application.

Claims Text - CLTX (9):

6. The system of claim 1 wherein the target application is an **object**-oriented application.

Claims Text - CLTX (21):

15. The method of claim 10 wherein the target application is an **object**-oriented application.

Claims Text - CLTX (50):
32. The system of claim 27 wherein the target application is an **object**-oriented application.

Claims Text - CLTX (61):
41. The method of claim 36 wherein the target application is an **object**-oriented application.

Other Reference Publication - OREF (3):
Berg, K.S., "Business **Objects** Done Right," Software Reviews, 20(4):175-176, (Apr. 1995).

US005960200A

United States Patent [19]

Eager et al.

[11] Patent Number: 5,960,200

[45] Date of Patent: Sep. 28, 1999

[54] **SYSTEM TO TRANSITION AN ENTERPRISE TO A DISTRIBUTED INFRASTRUCTURE**

[75] Inventors: **Timothy Eager**, Fullerton, Calif.; **Madhav Anand**, Cambridge, Mass.; **Edouard Aslanian**, Hermosa Beach, Calif.

[73] Assignee: **i-CUBE**, Cambridge, Mass.

[21] Appl. No.: **08/714,205**

[22] Filed: **Sep. 16, 1996**

Related U.S. Application Data

[60] Provisional application No. 60/016,330, May 3, 1996.

[51] **Int. Cl.**[6] **G06F 9/45**

[52] **U.S. Cl.** **395/705**; 395/701; 395/707; 395/500; 395/200.31; 705/7

[58] **Field of Search** 395/705, 701, 395/702, 707, 708, 500, 200.31, 200.33, 682, 683; 705/7–11; 707/10, 100, 102–104; 364/578, 468.02, 468.03, 468.05, 468.09

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,119,465	6/1992	Jack et al.	395/500
5,228,137	7/1993	Kleinerman et al.	395/500
5,455,948	10/1995	Poole et al.	707/102
5,457,797	10/1995	Butterworth et al.	395/682
5,524,253	6/1996	Pham et al.	395/200.32
5,606,697	2/1997	Ono	395/707

OTHER PUBLICATIONS

Callahan, J.R., et al., "A Packaging System For Heterogeneous Execution Environments," *IEEE Transactions on Software Engineering*, 17(6):626–635, (Jun. 1991).

Scandura, J.M., "Converting Legacy Code into Ada: A Cognitive Approach," *IEEE*, 27(4):55–61, (Apr. 1994).

Berg, K.S., "Business Objects Done Right," *Software Reviews*, 20(4):175–176, (Apr. 1995).

Ladd, D.A., et al., "A*: a Language for Implementing Language Processors," *IEEE*, pp. 1–10, (1994).

Leymann, F., et al., "Business Process Management With FlowMark," *IEEE*, pp. 230–234, (1994).

Tumminaro, J., Old School: 3R's New School: R/3, *InformationWeek*, pp. 50–54, (Apr. 1995).

Bartholomew, D., "SAP America's Trojan Horse," *Informationweek*, pp. 37–46, (Apr. 1995).

Tumminaro, J., "Forté Leads 3–Tier Pack," *Informationweek*, pp. 54–59, (May 1995).

Baum, D., "Three Tiers For Client–Server," *Informationweek*, pp. 42, 44, 48–49 and 52, (May 1995).

Maskell, K., "Building Software Bridges," *Systems International*, pp. 63–64, (Jan. 1987).

Ahuja, G.S., et al., "Role of Relational Data Base Management System An Client/Server Technology In EMS Migration," comprising 7 pages.

Moore, M., et al., "Knowledge–based User Interface Migration," *IEEE*, pp. 72–79, (1994).

Geschickter, C., "A Commonsense Plan for Client–Server Migration," *Data Communications*, pp. 73–76 and 78, (May 1994).

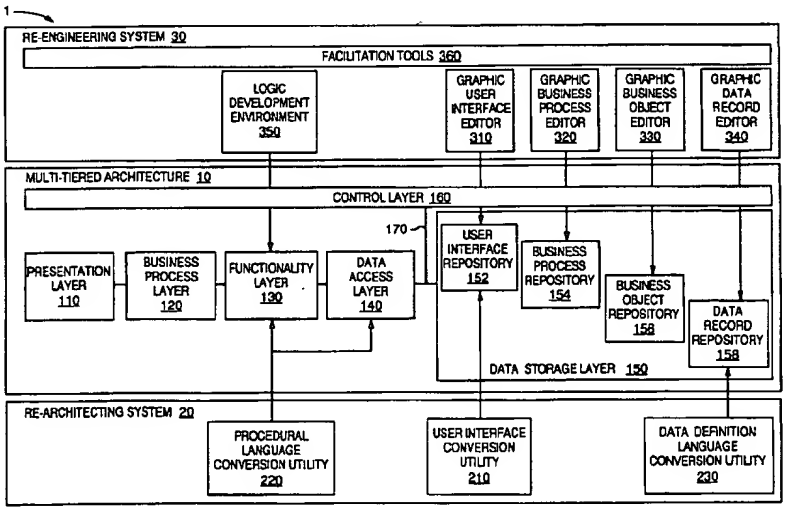
(List continued on next page.)

Primary Examiner—Tariq R. Hafiz
Assistant Examiner—Tuan Q. Dam
Attorney, Agent, or Firm—Hamilton, Brook, Smith & Reynolds, P.C.

[57] **ABSTRACT**

An automated system transitions an entire enterprise to a distributed infrastructure. The system includes a process for organizing and managing the transition, a multi-tiered client/server architecture that adheres to open systems standards, a system to automate the transition of existing applications to this architecture, and a system to enable the creation or modification of applications based on this architecture.

54 Claims, 36 Drawing Sheets



OTHER PUBLICATIONS

Mackey, S.R., et al., "Software Migration and Reengineering (SMR) A Pilot Project in Reengineering," pp. 178–191.
Mittra, S.S., "A Road Map for Migrating Legacy Systems to Client/Server," Software Maintenance: Research and Practice, vol. 7, pp. 117–130, (1995).
"DASE—Base Technology for Data and Applications Software Evolution and HIREL HIerarchical to RELational for an automated Migration to an open, relational and Client/Server positioned Environement," SWS Software Services, pp. 1–4, (1994).
"DASE Base Technology for Data and Acquisitions Software Evolution and HIREL/AProp IMS/DB–DB2 dual way Access Propagation for an automated Migration to an open, relational and Client/Server positioned Environment," SWS Software Services, pp. 1–4, (1994).
"DASE Base Technology for Data and Applications Software Evolution and IXREL IBM hIerarchisch to uniX RELational for an automated Migration to an open, relational and Client/Server positioned Environment," SWS Software Services, pp. 1–4, (1994).
"DASE Base Technology for Data and Applications Software Evolution and CMP COBOL Migration Products for an automated Migration from COBOL ANS'68/74 to COBOL ANS'85 and REPORT WRITER to Native COBOL," SWS Software Services, pp. 1–4, (1994).
"DASE Base Technology for Data and Acquisitions Software Evolution and VREL Vsam to RELational for an automated Migration to an open, relational and Client/Server positioned Environment," SWS Software Services, pp. 1–4, (1994).

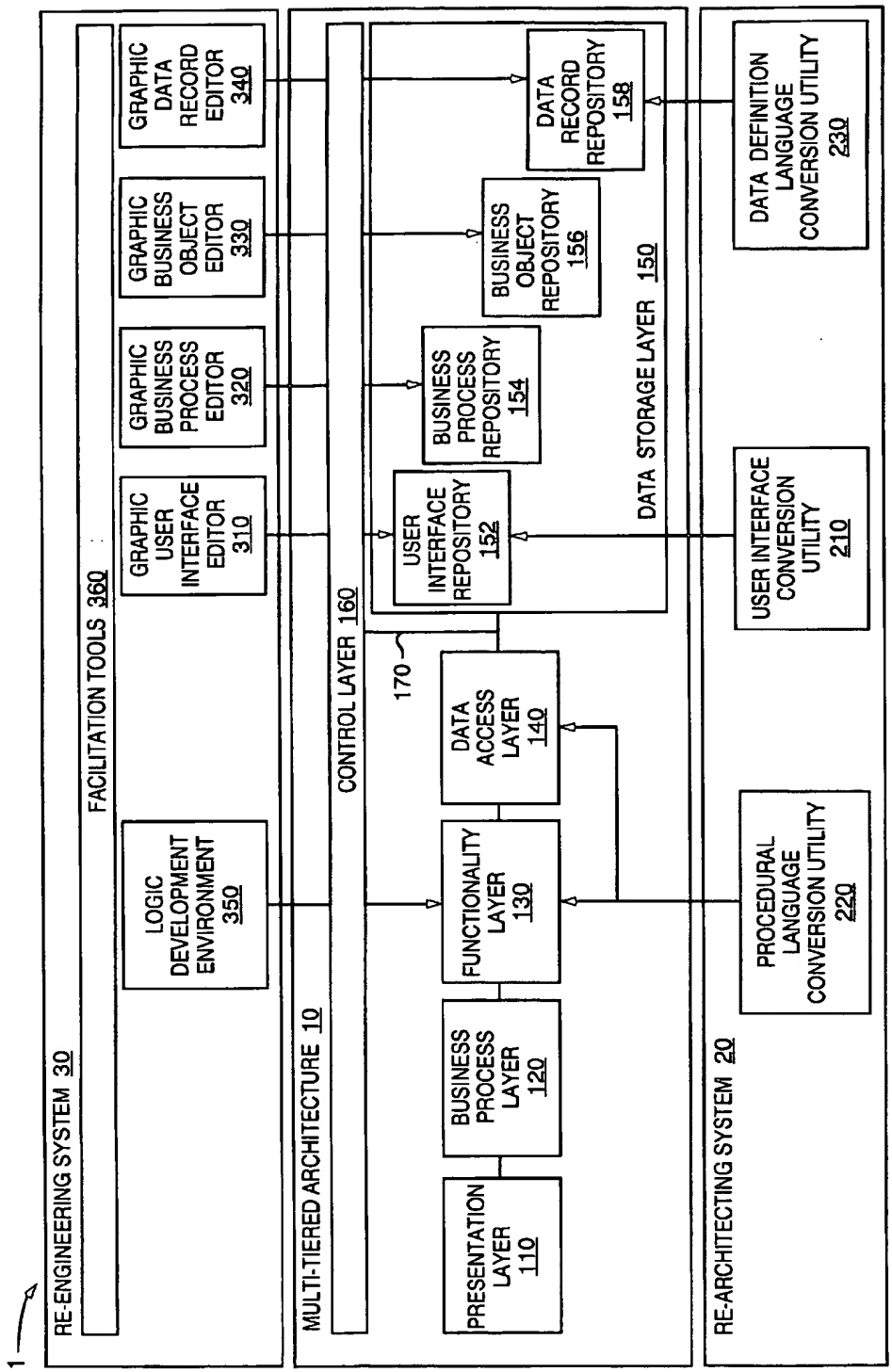

1
RE-ENGINEERING SYSTEM 30
FACILITATION TOOLS 360
GRAPHIC DATA RECORD EDITOR 340
GRAPHIC BUSINESS OBJECT EDITOR 330
GRAPHIC BUSINESS PROCESS EDITOR 320
GRAPHIC USER INTERFACE EDITOR 310
LOGIC DEVELOPMENT ENVIRONMENT 350
MULTI-TIERED ARCHITECTURE 10
CONTROL LAYER 160
170
PRESENTATION LAYER 110
BUSINESS PROCESS LAYER 120
FUNCTIONALITY LAYER 130
DATA ACCESS LAYER 140
DATA STORAGE LAYER 150
USER INTERFACE REPOSITORY 152
BUSINESS PROCESS REPOSITORY 154
BUSINESS OBJECT REPOSITORY 156
DATA RECORD REPOSITORY 158
RE-ARCHITECTING SYSTEM 20
PROCEDURAL LANGUAGE CONVERSION UTILITY 220
USER INTERFACE CONVERSION UTILITY 210
DATA DEFINITION LANGUAGE CONVERSION UTILITY 230

FIG. 1

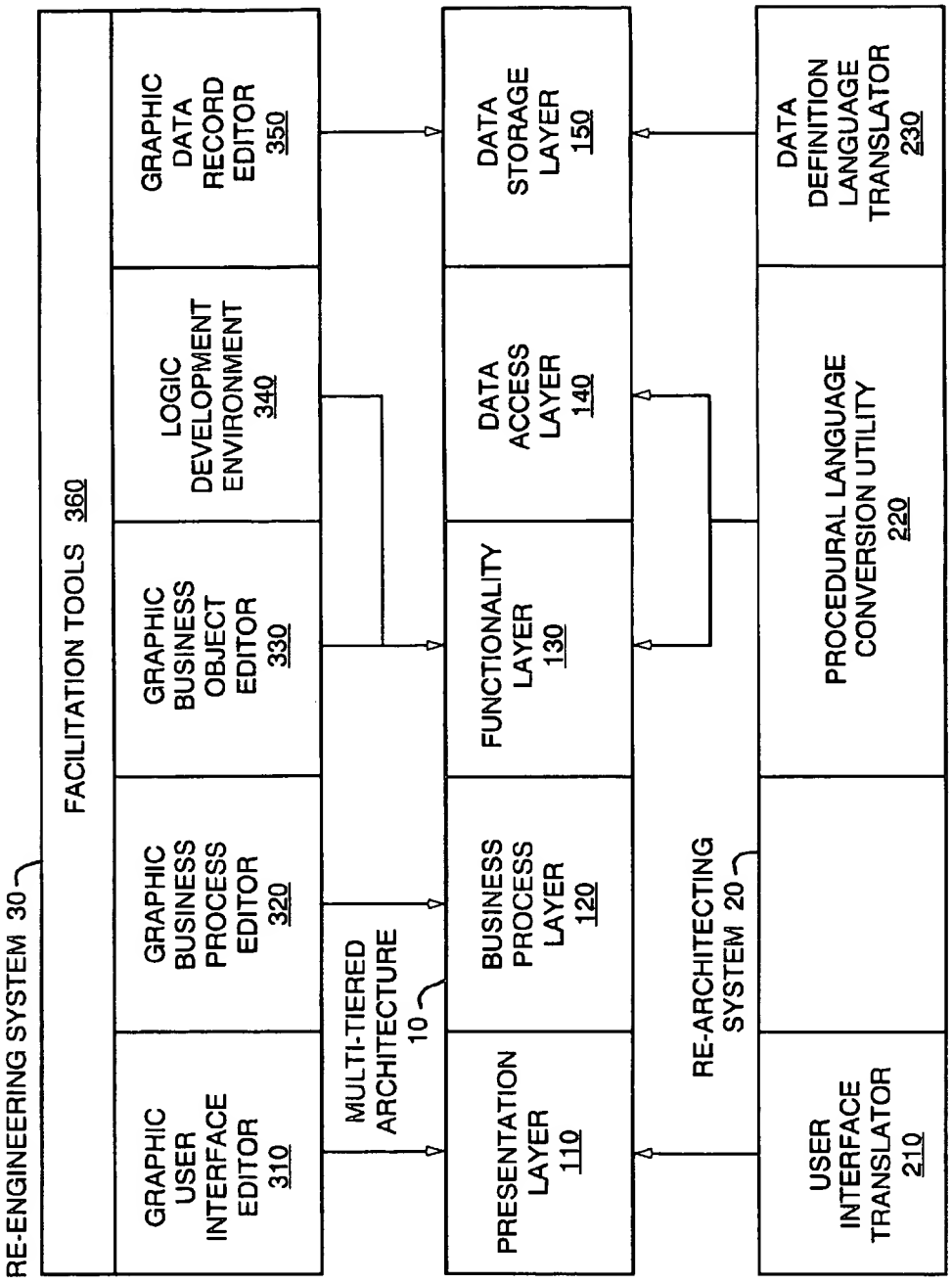

RE-ENGINEERING SYSTEM 30
FACILITATION TOOLS 360
GRAPHIC USER INTERFACE EDITOR 310
GRAPHIC BUSINESS PROCESS EDITOR 320
GRAPHIC BUSINESS OBJECT EDITOR 330
LOGIC DEVELOPMENT ENVIRONMENT 340
GRAPHIC DATA RECORD EDITOR 350
MULTI-TIERED ARCHITECTURE 10
PRESENTATION LAYER 110
BUSINESS PROCESS LAYER 120
FUNCTIONALITY LAYER 130
DATA ACCESS LAYER 140
DATA STORAGE LAYER 150
RE-ARCHITECTING SYSTEM 20
USER INTERFACE TRANSLATOR 210
PROCEDURAL LANGUAGE CONVERSION UTILITY 220
DATA DEFINITION LANGUAGE TRANSLATOR 230

FIG. 2

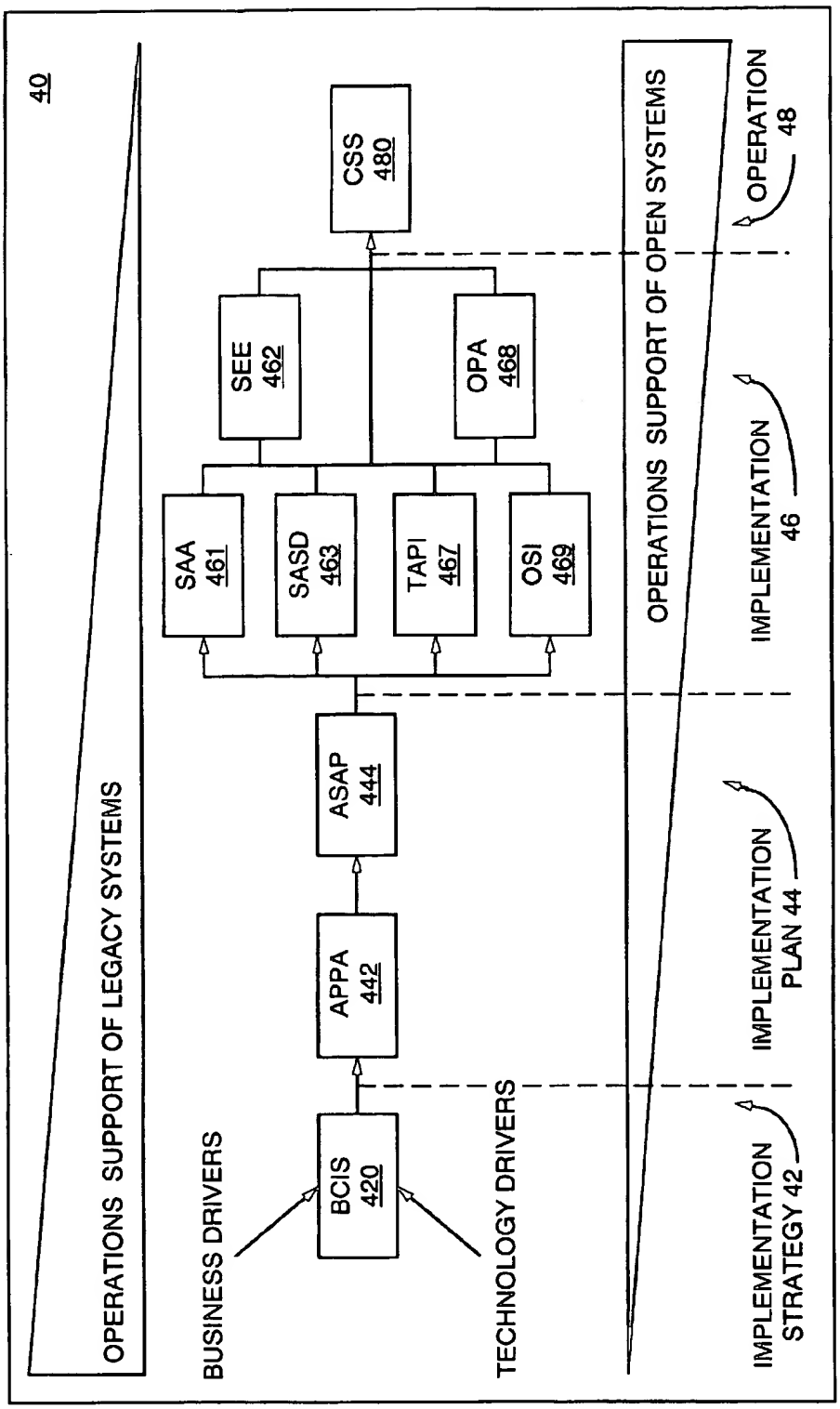

40
OPERATIONS SUPPORT OF LEGACY SYSTEMS
OPERATIONS SUPPORT OF OPEN SYSTEMS
BUSINESS DRIVERS
TECHNOLOGY DRIVERS
BCIS 420
APPA 442
ASAP 444
SAA 461
SASD 463
TAPI 467
OSI 469
SEE 462
OPA 468
CSS 480
IMPLEMENTATION STRATEGY 42
IMPLEMENTATION PLAN 44
IMPLEMENTATION 46
OPERATION 48

FIG. 3

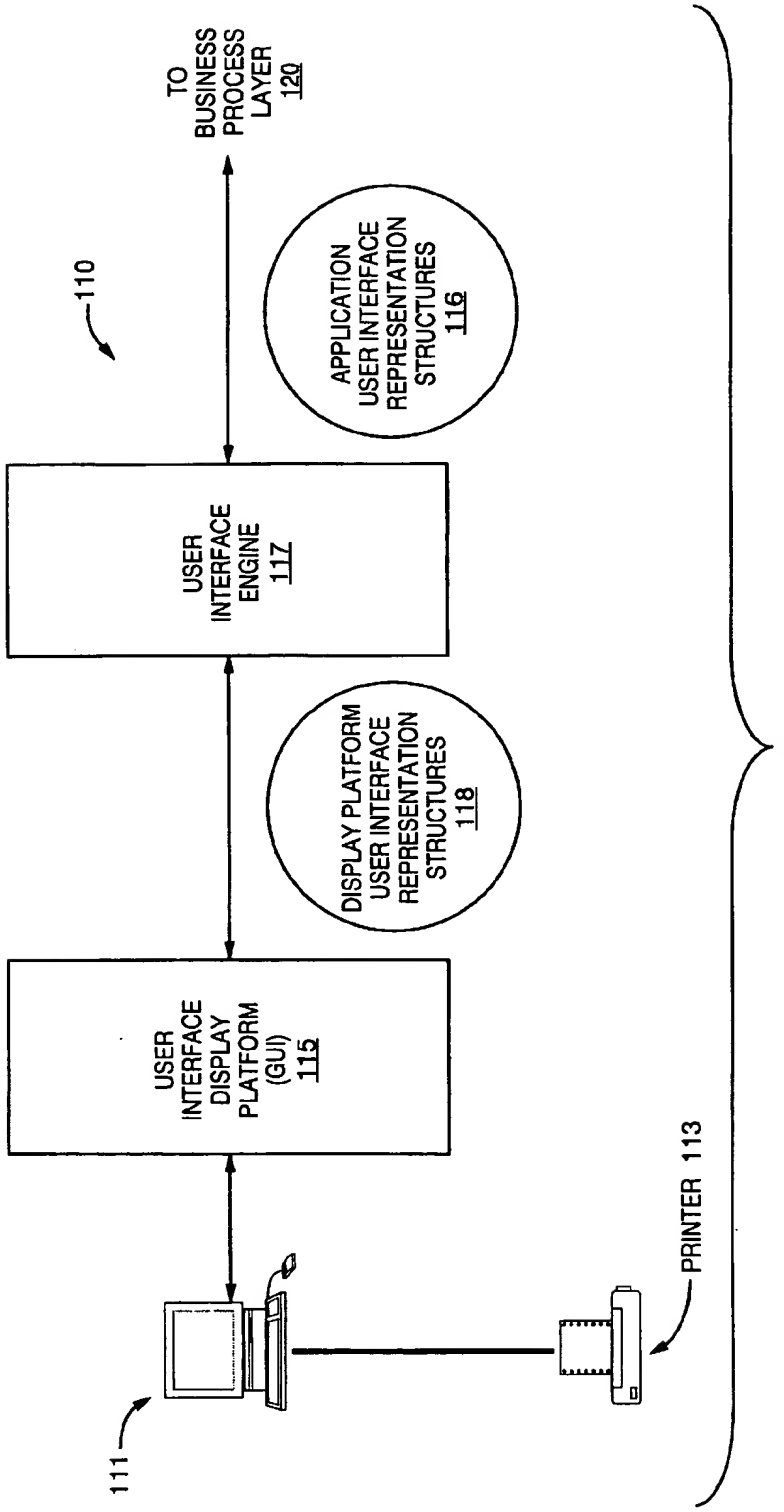

TO BUSINESS PROCESS LAYER 120
110
APPLICATION USER INTERFACE REPRESENTATION STRUCTURES 116
USER INTERFACE ENGINE 117
DISPLAY PLATFORM USER INTERFACE REPRESENTATION STRUCTURES 118
USER INTERFACE DISPLAY PLATFORM (GUI) 115
PRINTER 113
111

FIG. 4

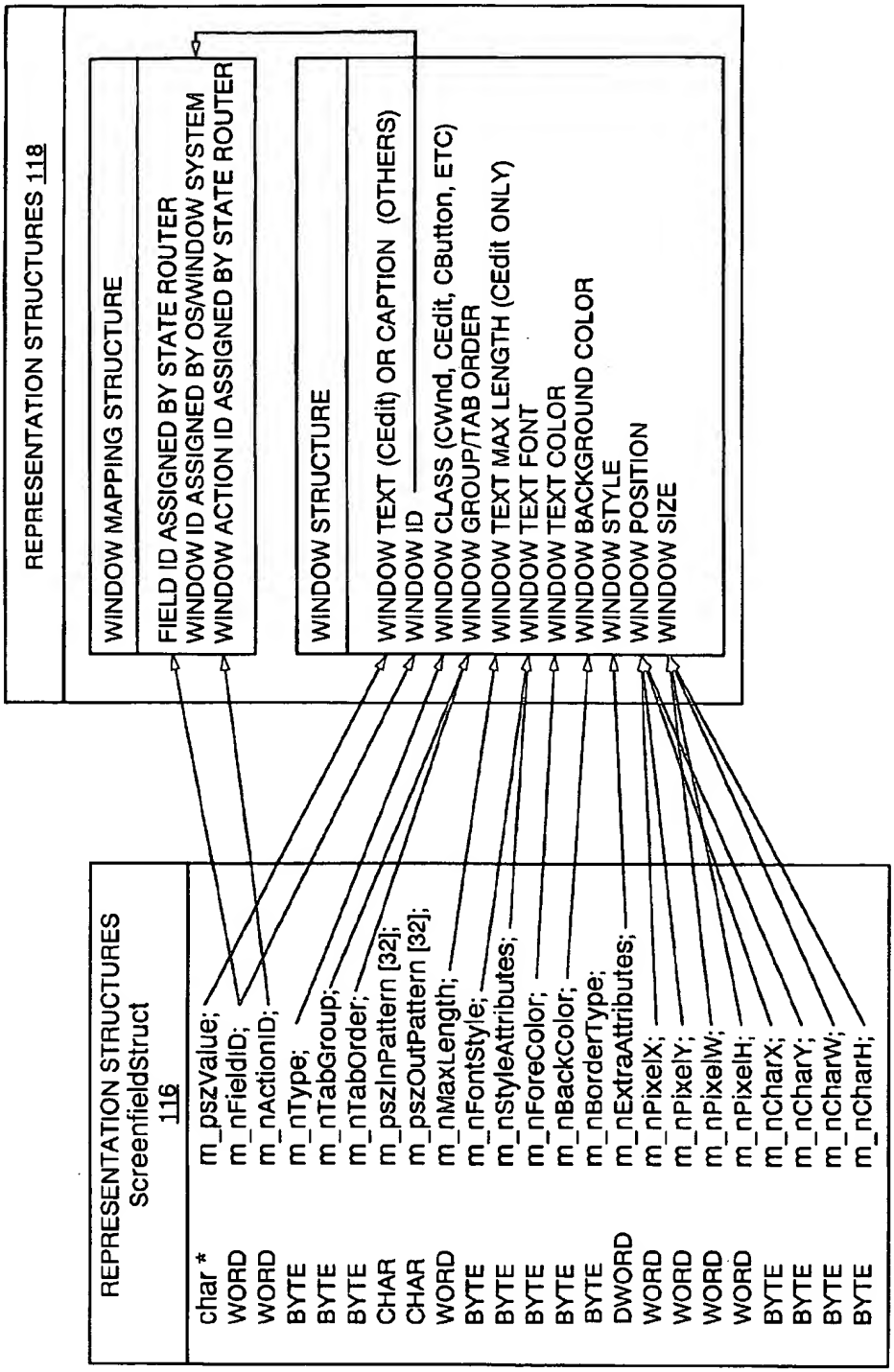

REPRESENTATION STRUCTURES 118
WINDOW MAPPING STRUCTURE
FIELD ID ASSIGNED BY STATE ROUTER
WINDOW ID ASSIGNED BY OS/WINDOW SYSTEM
WINDOW ACTION ID ASSIGNED BY STATE ROUTER
WINDOW STRUCTURE
WINDOW TEXT (CEdit) OR CAPTION (OTHERS)
WINDOW ID
WINDOW CLASS (CWnd, CEdit, CButton, ETC)
WINDOW GROUP/TAB ORDER
WINDOW TEXT MAX LENGTH (CEdit ONLY)
WINDOW TEXT FONT
WINDOW TEXT COLOR
WINDOW BACKGROUND COLOR
WINDOW STYLE
WINDOW POSITION
WINDOW SIZE
REPRESENTATION STRUCTURES
ScreenfieldStruct
116
char* m_pszValue;
WORD m_nFieldID;
WORD m_nActionID;
BYTE m_nType;
BYTE m_nTabGroup;
BYTE m_nTabOrder;
CHAR m_pszInPattern [32];
CHAR m_pszOutPattern [32];
WORD m_nMaxLength;
BYTE m_nFontStyle;
BYTE m_nStyleAttributes;
BYTE m_nForeColor;
BYTE m_nBackColor;
BYTE m_nBorderType;
DWORD m_nExtraAttributes;
WORD m_nPixelX;
WORD m_nPixelY;
WORD m_nPixelW;
WORD m_nPixelH;
BYTE m_nCharX;
BYTE m_nCharY;
BYTE m_nCharW;
BYTE m_nCharH;

FIG. 5

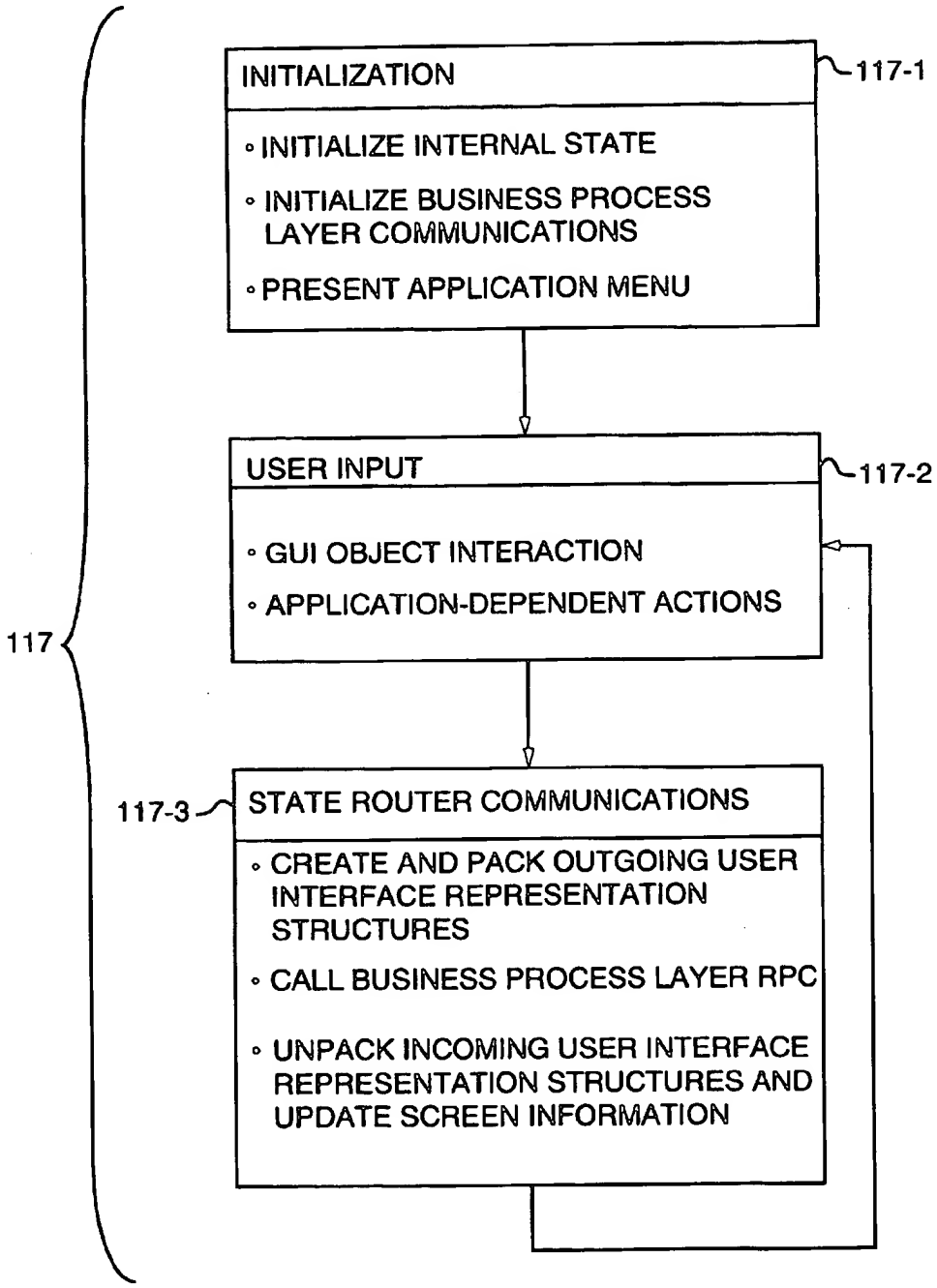

INITIALIZATION
117-1
◦ INITIALIZE INTERNAL STATE
◦ INITIALIZE BUSINESS PROCESS LAYER COMMUNICATIONS
◦ PRESENT APPLICATION MENU
USER INPUT
117-2
◦ GUI OBJECT INTERACTION
◦ APPLICATION-DEPENDENT ACTIONS
117
117-3
STATE ROUTER COMMUNICATIONS
◦ CREATE AND PACK OUTGOING USER INTERFACE REPRESENTATION STRUCTURES
◦ CALL BUSINESS PROCESS LAYER RPC
◦ UNPACK INCOMING USER INTERFACE REPRESENTATION STRUCTURES AND UPDATE SCREEN INFORMATION

FIG. 6

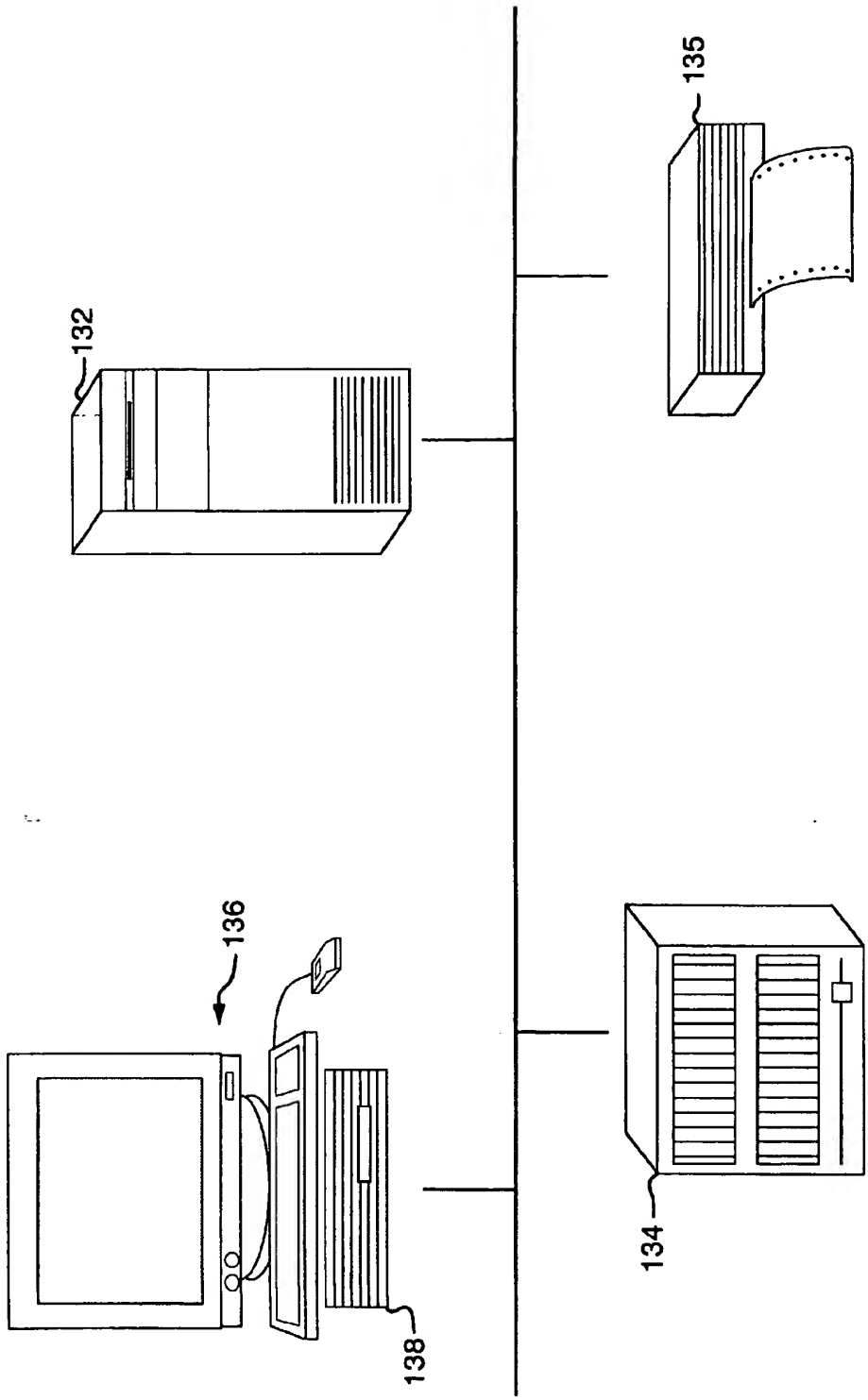


FIG. 7

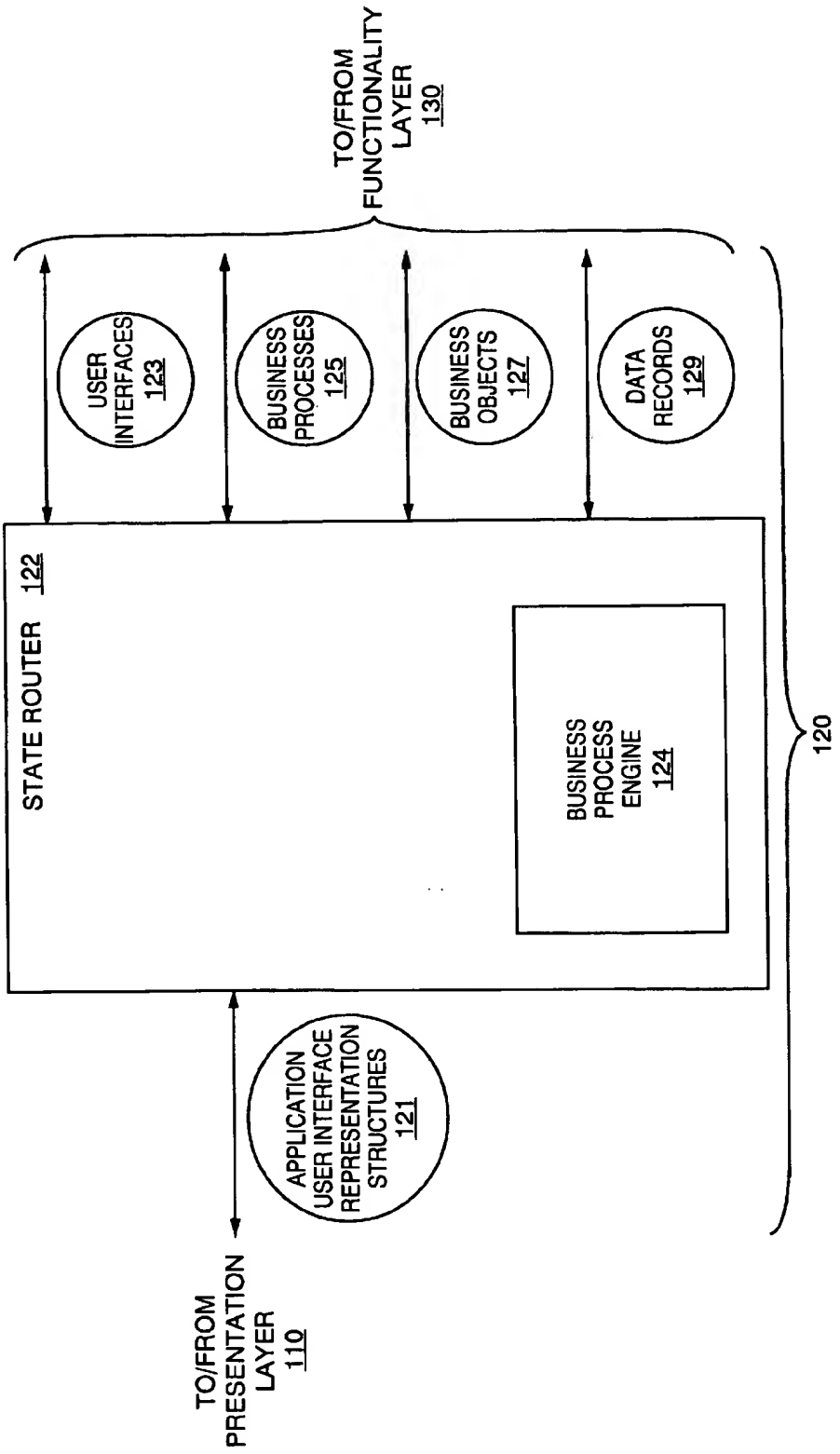

TO/FROM FUNCTIONALITY LAYER 130
USER INTERFACES 123
BUSINESS PROCESSES 125
BUSINESS OBJECTS 127
DATA RECORDS 129
STATE ROUTER 122
BUSINESS PROCESS ENGINE 124
120
APPLICATION USER INTERFACE REPRESENTATION STRUCTURES 121
TO/FROM PRESENTATION LAYER 110

FIG. 8

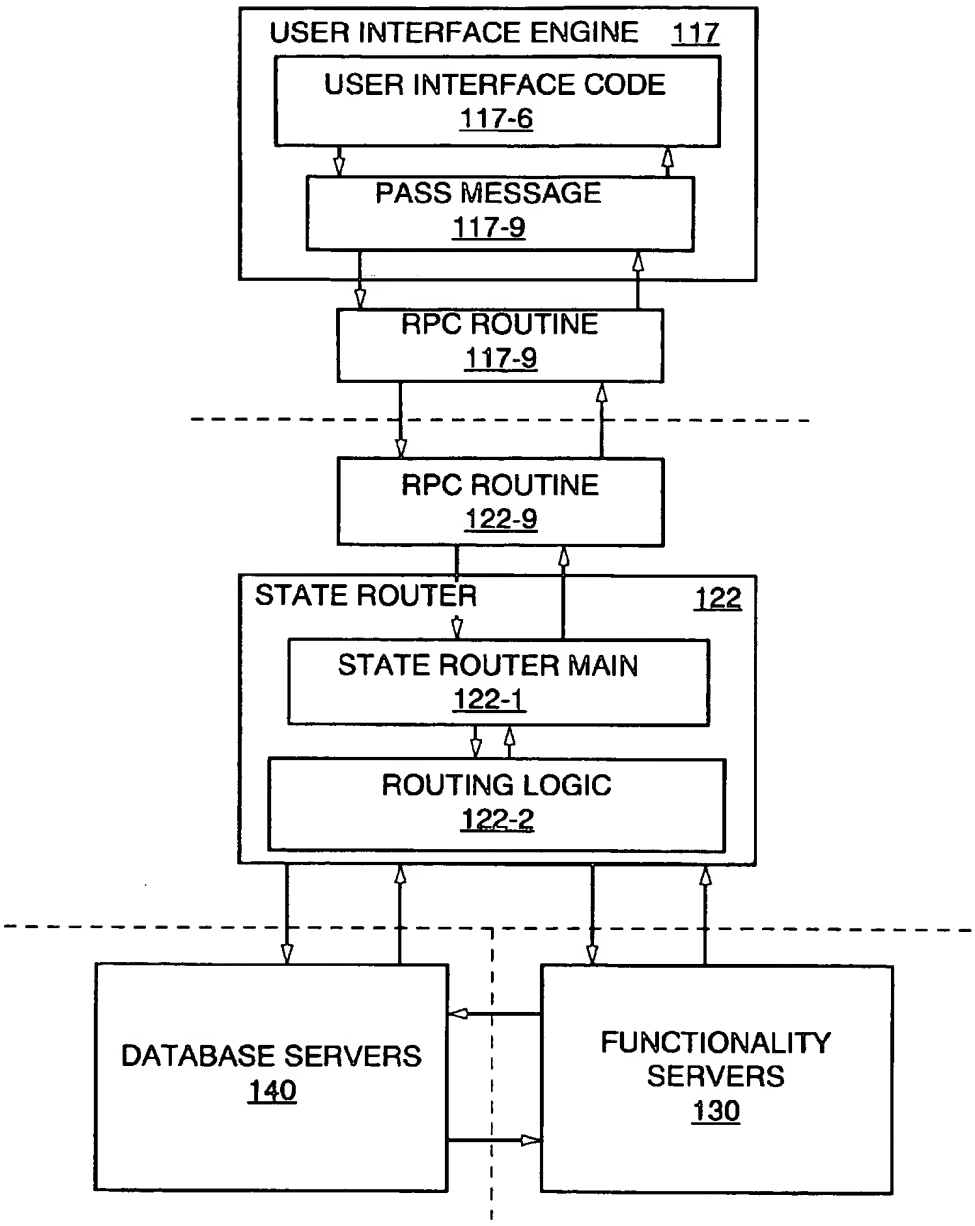

USER INTERFACE ENGINE 117
USER INTERFACE CODE
117-6
PASS MESSAGE
117-9
RPC ROUTINE
117-9
RPC ROUTINE
122-9
STATE ROUTER
122
STATE ROUTER MAIN
122-1
ROUTING LOGIC
122-2
DATABASE SERVERS
140
FUNCTIONALITY
SERVERS
130

FIG. 9

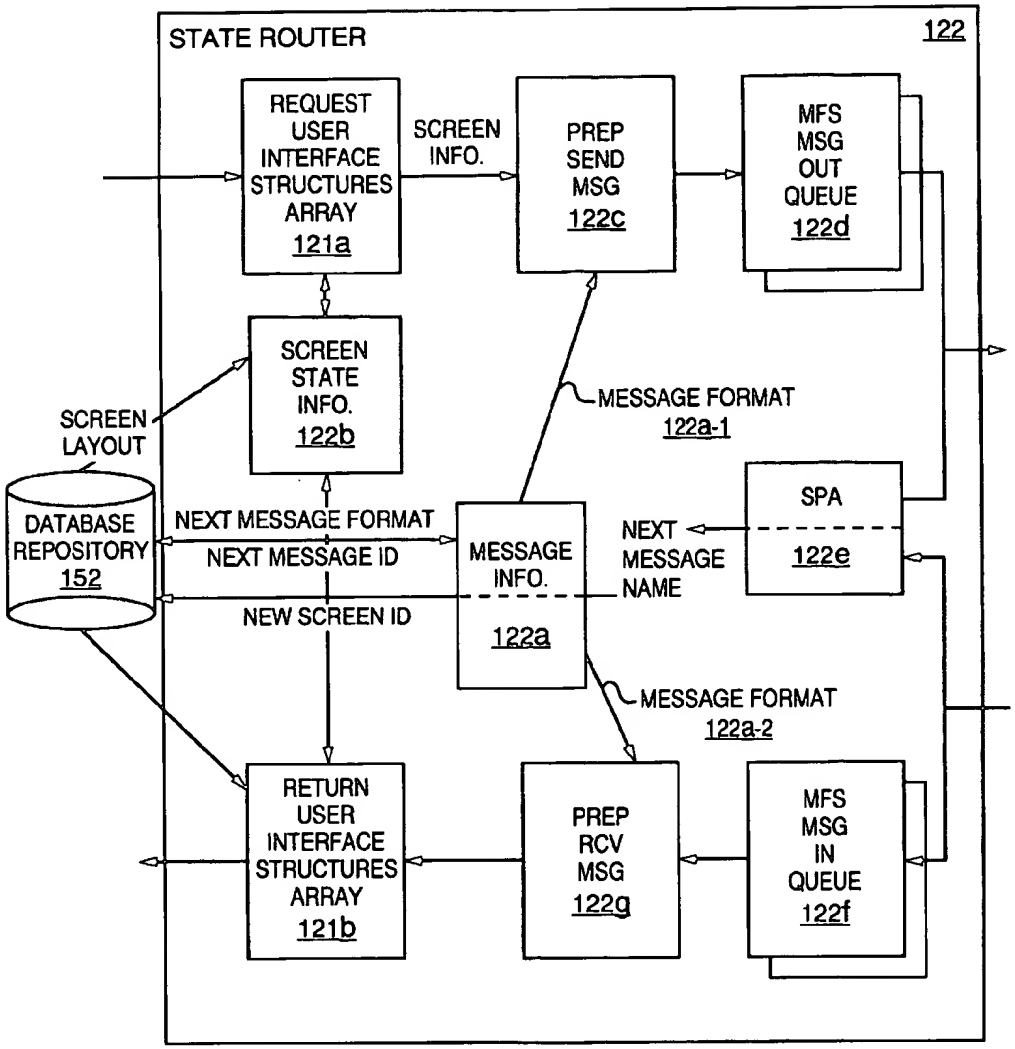

STATE ROUTER
122
REQUEST USER INTERFACE STRUCTURES ARRAY 121a
SCREEN INFO.
PREP SEND MSG 122c
MFS MSG OUT QUEUE 122d
SCREEN STATE INFO. 122b
SCREEN LAYOUT
MESSAGE FORMAT 122a-1
DATABASE REPOSITORY 152
NEXT MESSAGE FORMAT
NEXT MESSAGE ID
NEW SCREEN ID
MESSAGE INFO. 122a
SPA 122e
NEXT MESSAGE NAME
MESSAGE FORMAT 122a-2
RETURN USER INTERFACE STRUCTURES ARRAY 121b
PREP RCV MSG 122g
MFS MSG IN QUEUE 122f

FIG. 10

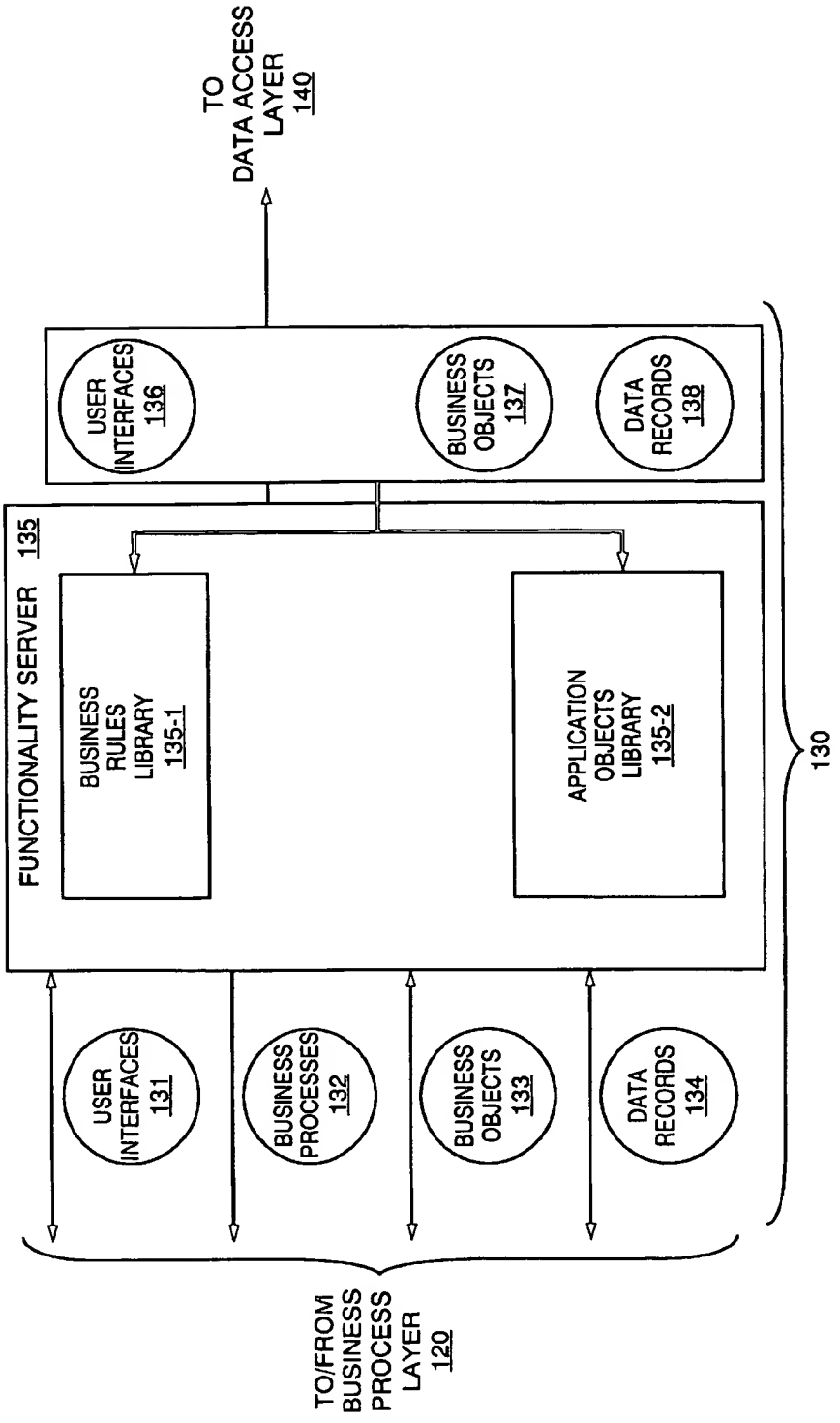

TO DATA ACCESS LAYER 140
USER INTERFACES 136
BUSINESS OBJECTS 137
DATA RECORDS 138
FUNCTIONALITY SERVER 135
BUSINESS RULES LIBRARY 135-1
APPLICATION OBJECTS LIBRARY 135-2
130
USER INTERFACES 131
BUSINESS PROCESSES 132
BUSINESS OBJECTS 133
DATA RECORDS 134
TO/FROM BUSINESS PROCESS LAYER 120

FIG. 11

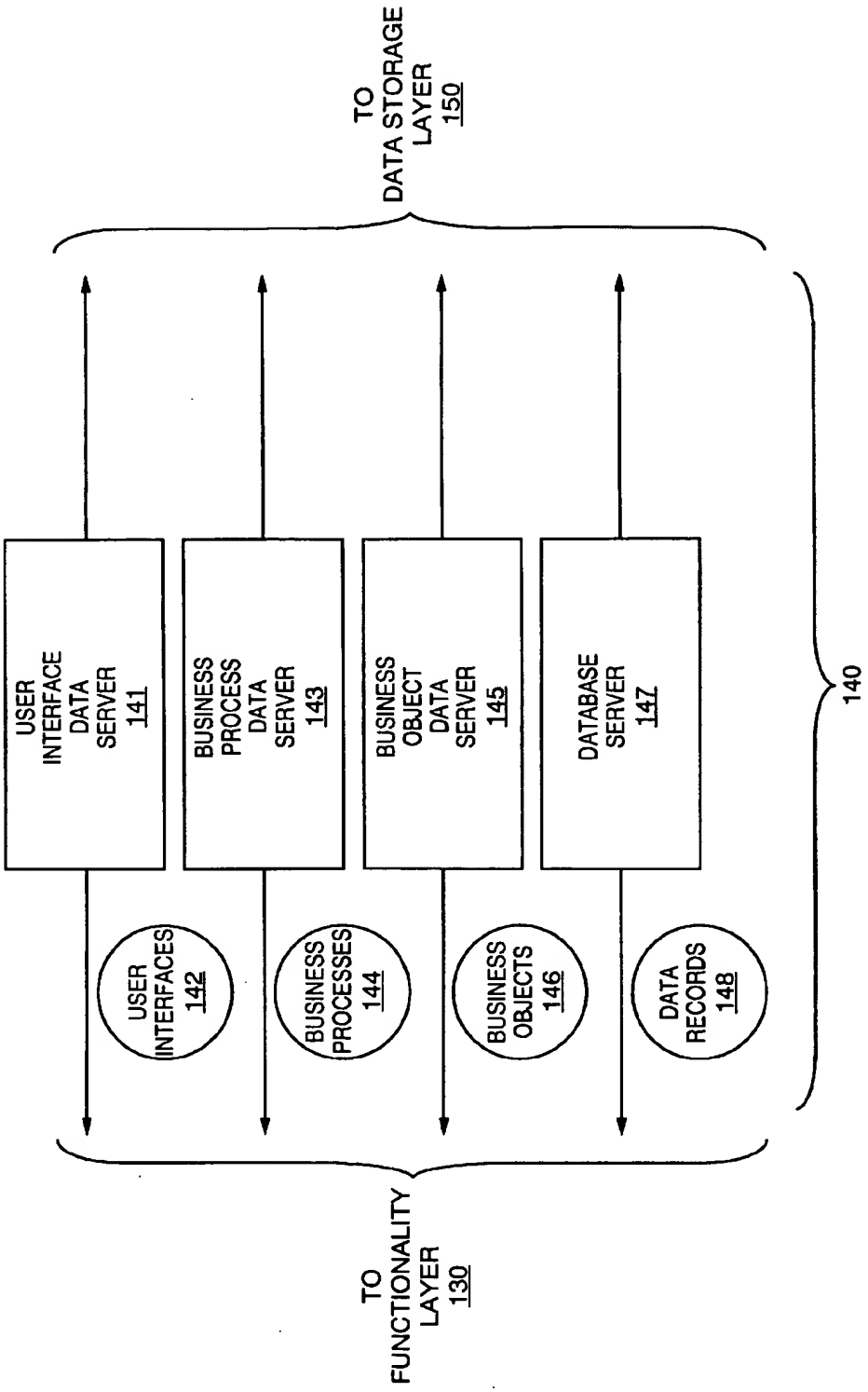

TO DATA STORAGE LAYER 150
USER INTERFACE DATA SERVER 141
BUSINESS PROCESS DATA SERVER 143
BUSINESS OBJECT DATA SERVER 145
DATABASE SERVER 147
140
USER INTERFACES 142
BUSINESS PROCESSES 144
BUSINESS OBJECTS 146
DATA RECORDS 148
TO FUNCTIONALITY LAYER 130

FIG. 12

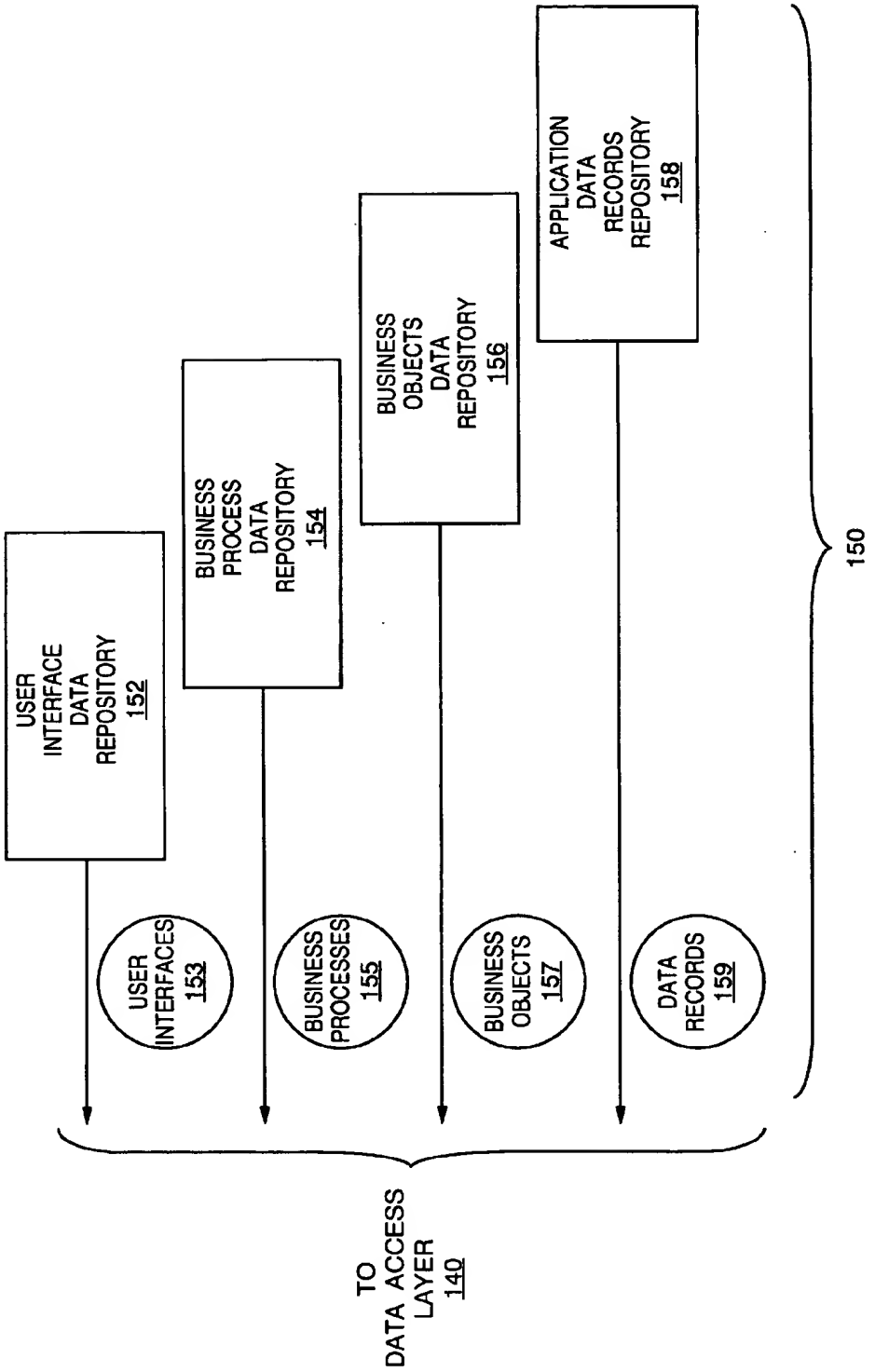

USER INTERFACE DATA REPOSITORY 152
BUSINESS PROCESS DATA REPOSITORY 154
BUSINESS OBJECTS DATA REPOSITORY 156
APPLICATION DATA RECORDS REPOSITORY 158
150
USER INTERFACES 153
BUSINESS PROCESSES 155
BUSINESS OBJECTS 157
DATA RECORDS 159
TO DATA ACCESS LAYER 140

FIG. 13

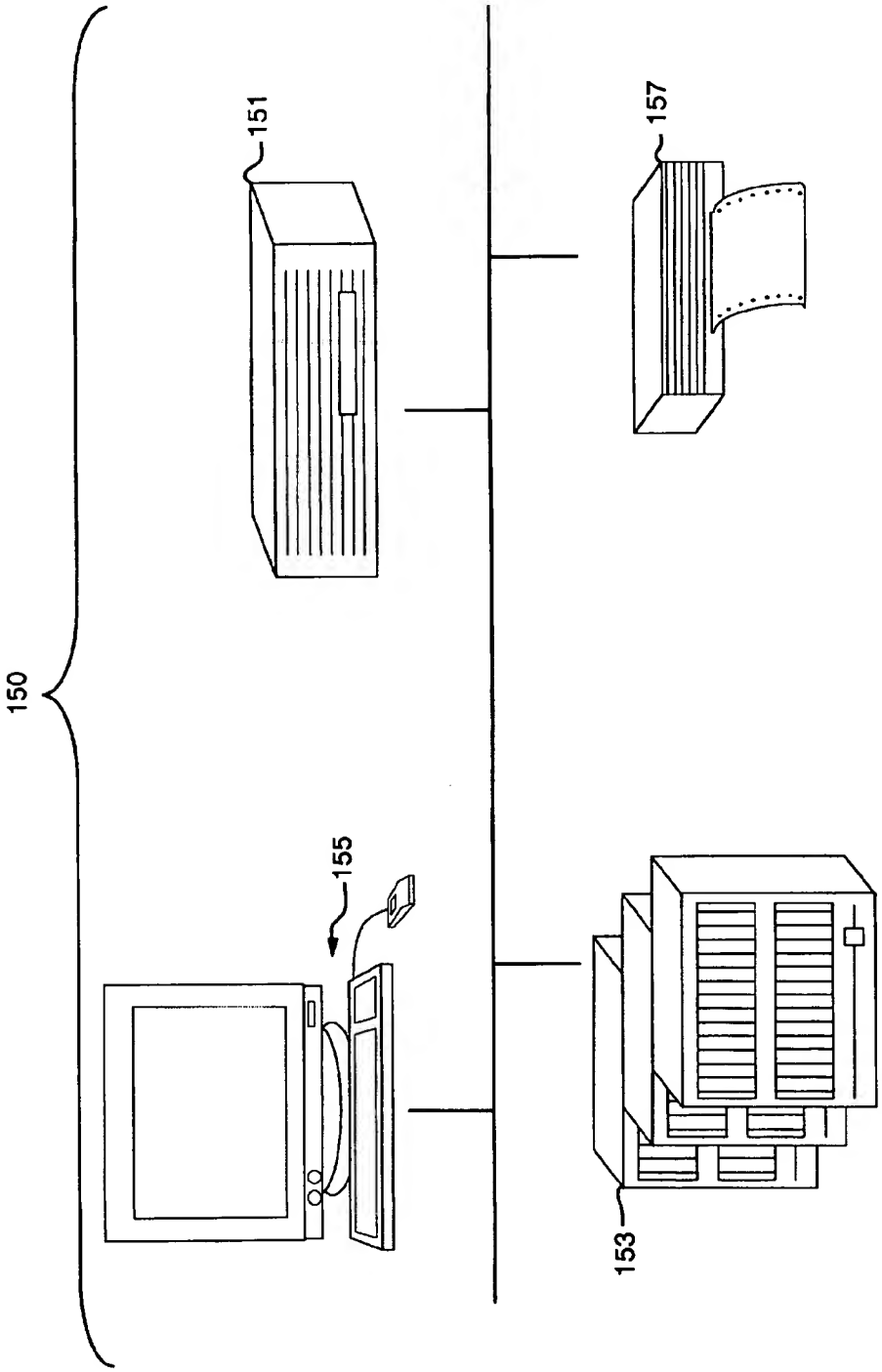


FIG. 14

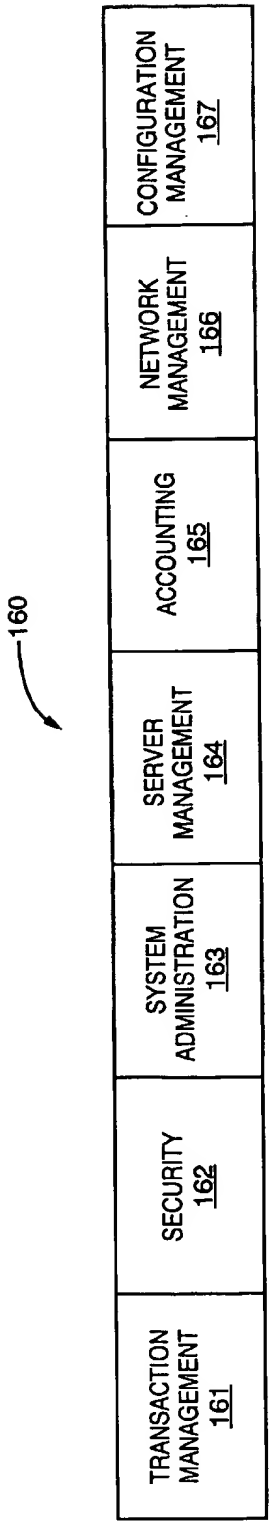

160
TRANSACTION MANAGEMENT 161
SECURITY 162
SYSTEM ADMINISTRATION 163
SERVER MANAGEMENT 164
ACCOUNTING 165
NETWORK MANAGEMENT 166
CONFIGURATION MANAGEMENT 167

FIG. 15

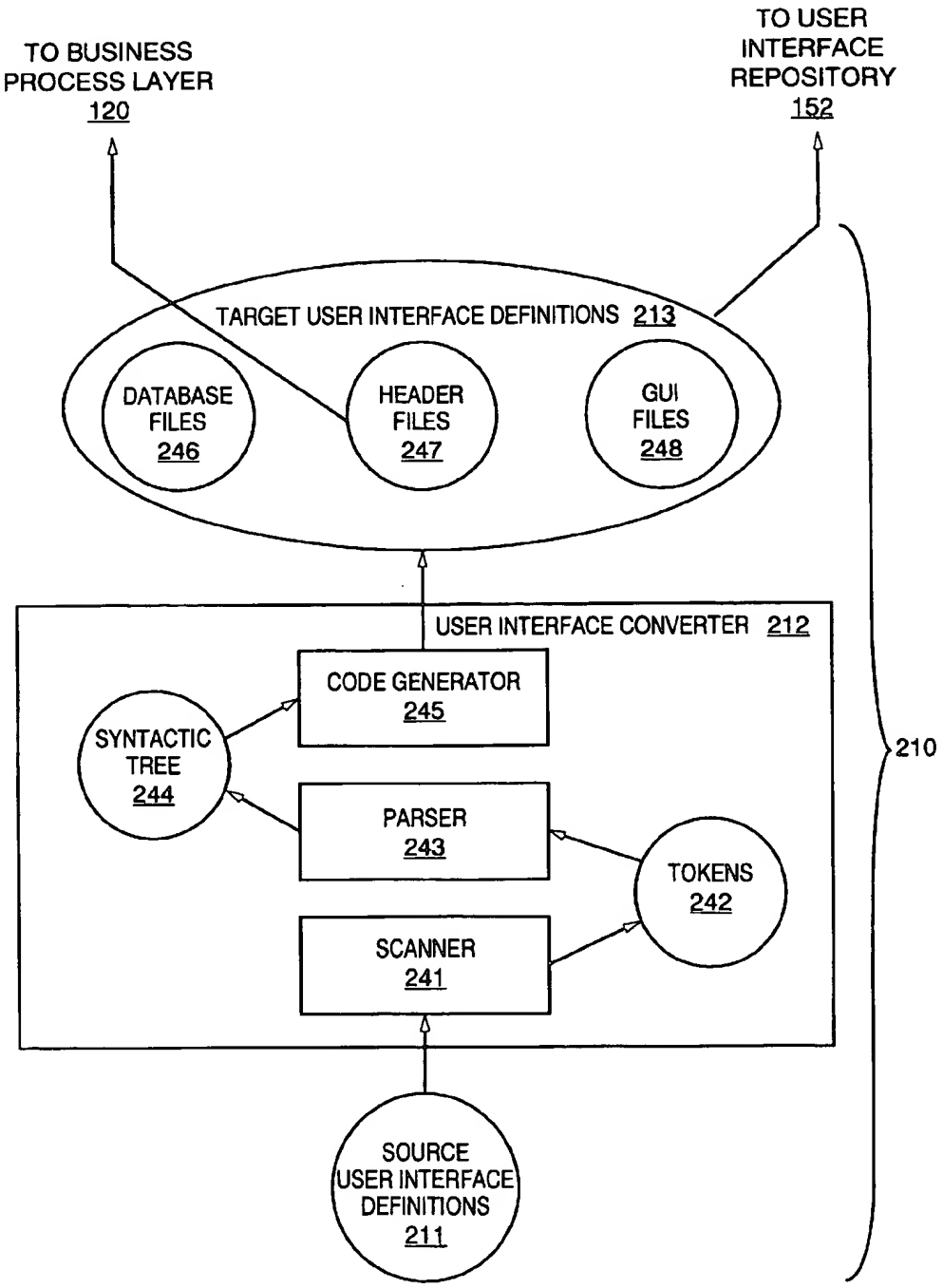

TO BUSINESS PROCESS LAYER 120
TO USER INTERFACE REPOSITORY 152
TARGET USER INTERFACE DEFINITIONS 213
DATABASE FILES 246
HEADER FILES 247
GUI FILES 248
USER INTERFACE CONVERTER 212
CODE GENERATOR 245
SYNTACTIC TREE 244
PARSER 243
TOKENS 242
SCANNER 241
SOURCE USER INTERFACE DEFINITIONS 211
210

FIG. 16

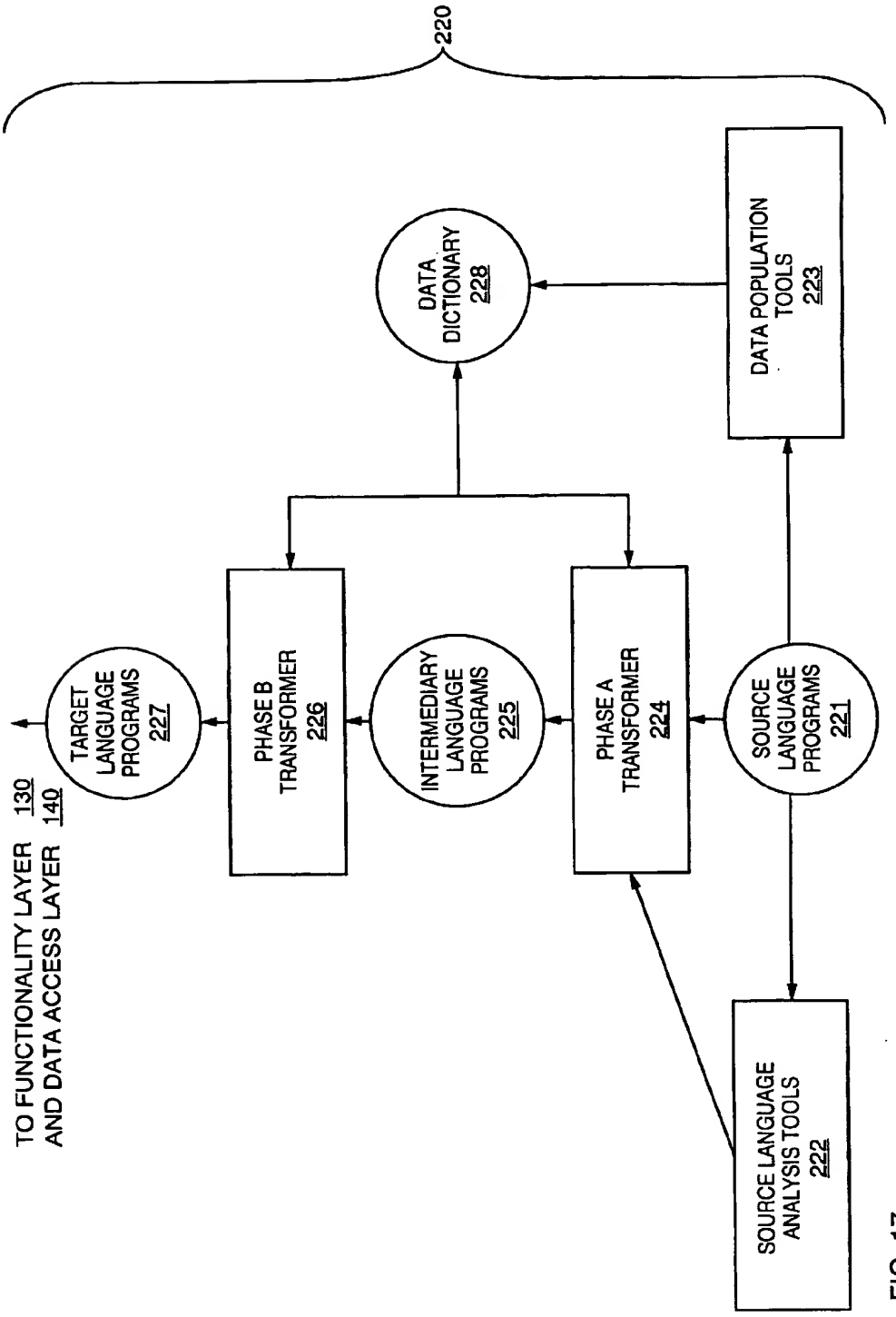
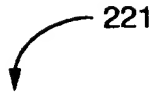

220
DATA DICTIONARY 228
DATA POPULATION TOOLS 223
TARGET LANGUAGE PROGRAMS 227
PHASE B TRANSFORMER 226
INTERMEDIARY LANGUAGE PROGRAMS 225
PHASE A TRANSFORMER 224
SOURCE LANGUAGE PROGRAMS 221
SOURCE LANGUAGE ANALYSIS TOOLS 222
TO FUNCTIONALITY LAYER 130
AND DATA ACCESS LAYER 140

FIG. 17




```
MOVE LOW-VALUE TO FLAG-AREA YMS-CALL_RESULTS. 221-A
IF YMS-CALL-AREA > 9000 221-B
  MOVE 'N' TO YMS-CALL-ERROR 221-C
  GO TO 0099-FIN. 221-D
MOVE 0 TO BIT-NUM. 221-E
MOVE YMS-CALL-VT TO HOLD-CALL-VT. 221-F
IF HOLD-CALL-VT-2-NUM = 0 221-G
  MOVE SPACE TO HOLD-CALL-VT-2-CHAR. 221-H
PERFORM 1000-FIND-COMMON-BANK. 221-I
IF NOT END-PROGRAM 221-J
  IF YMS-CALL-ERROR > SPACE 221-K
    NEXT SENTENCE 221-L
  ELSE 221-M
    PERFORM 2000-SEARCH-INDEX. 221-N
0099-FIN. 221-O
```

FIG. 18

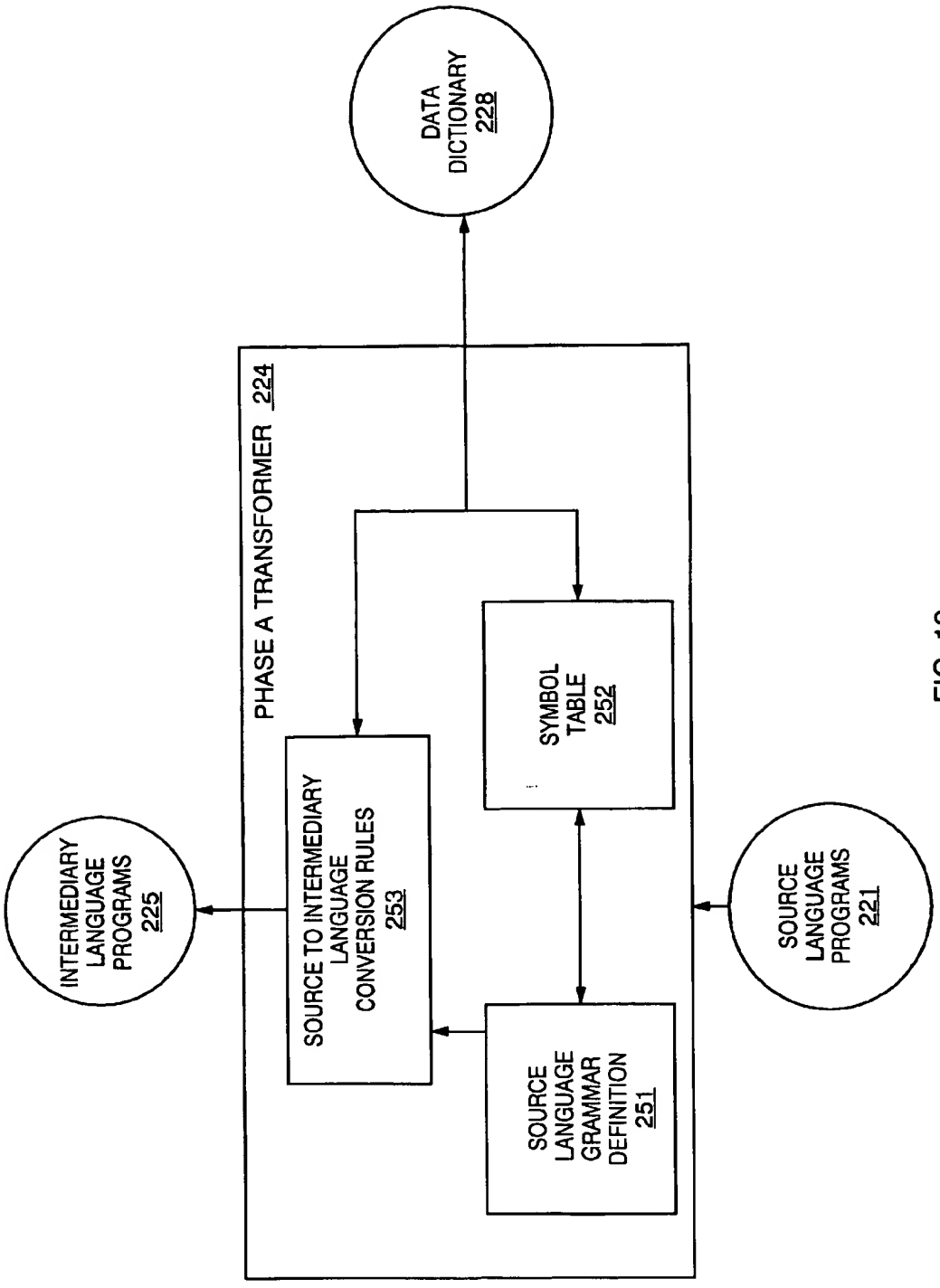

DATA DICTIONARY 228
PHASE A TRANSFORMER 224
SYMBOL TABLE 252
SOURCE TO INTERMEDIARY LANGUAGE CONVERSION RULES 253
SOURCE LANGUAGE GRAMMAR DEFINITION 251
INTERMEDIARY LANGUAGE PROGRAMS 225
SOURCE LANGUAGE PROGRAMS 221

FIG. 19

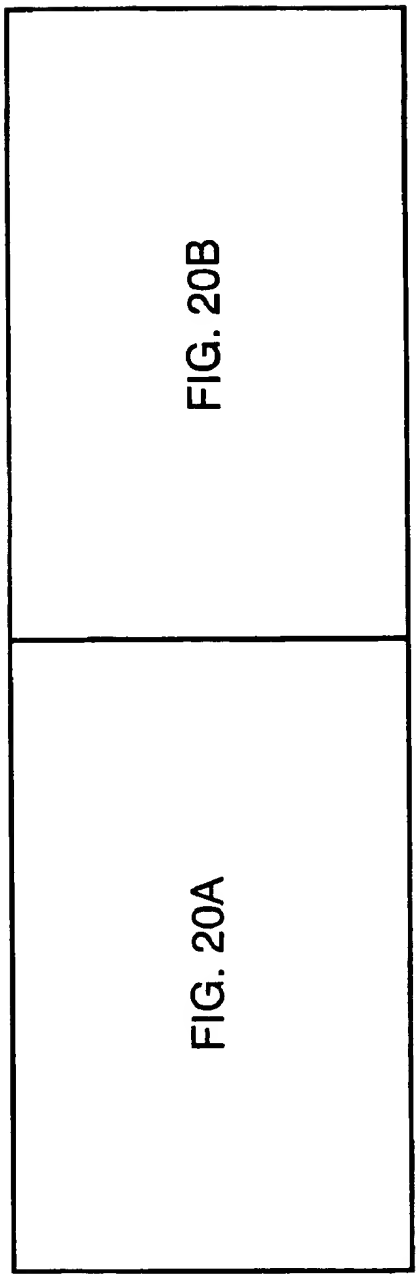


FIG. 20

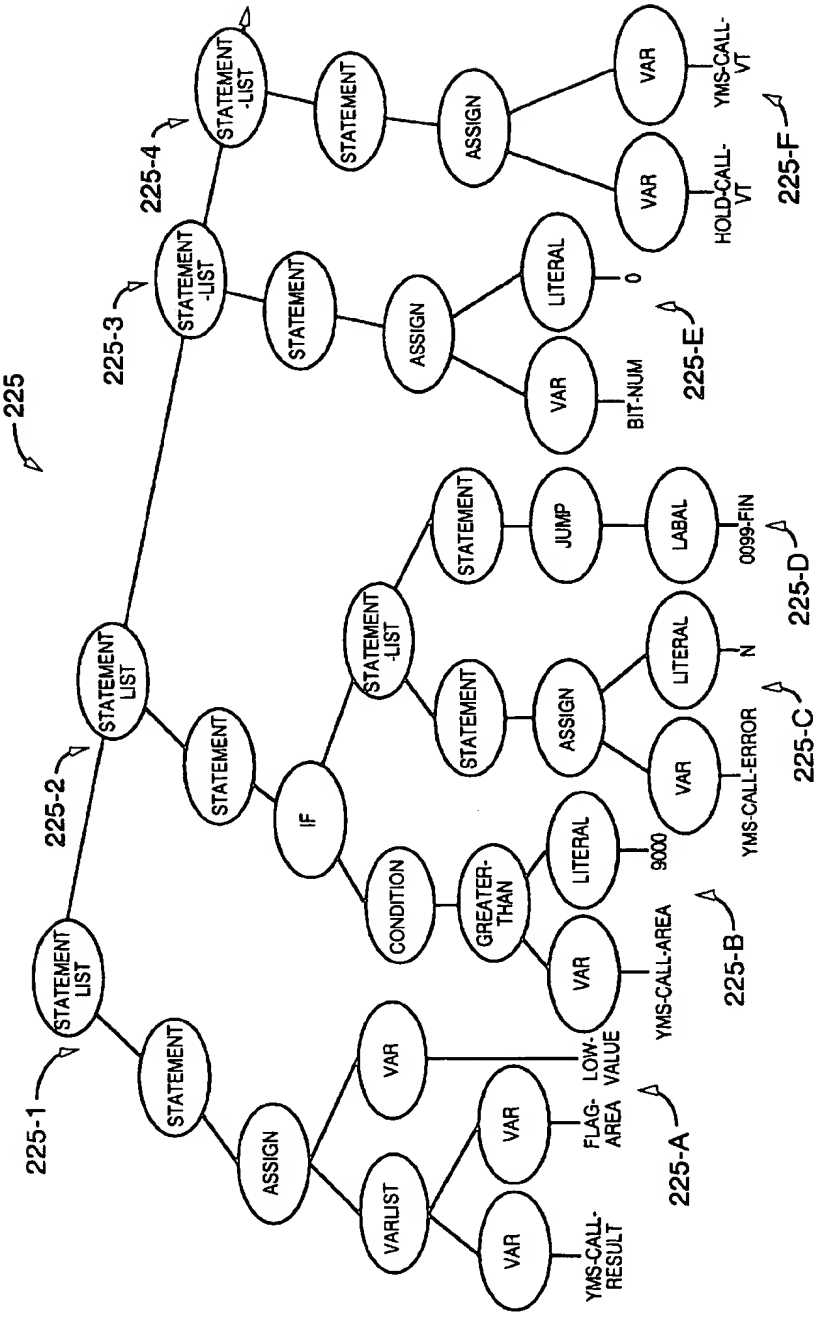

225
225-1
225-2
225-3
225-4
225-A
225-B
225-C
225-D
225-E
225-F
STATEMENT LIST
STATEMENT
ASSIGN
VAR
VARLIST
VAR
VAR
YMS-CALL-RESULT
FLAG-AREA
LOW-VALUE
IF
CONDITION
GREATER-THAN
VAR
YMS-CALL-AREA
LITERAL
9000
STATEMENT-LIST
STATEMENT
ASSIGN
VAR
YMS-CALL-ERROR
LITERAL
N
STATEMENT
JUMP
LABAL
0099-FIN
STATEMENT-LIST
STATEMENT
ASSIGN
VAR
BIT-NUM
LITERAL
0
STATEMENT-LIST
STATEMENT
ASSIGN
VAR
HOLD-CALL-VT
VAR
YMS-CALL-VT

FIG. 20A

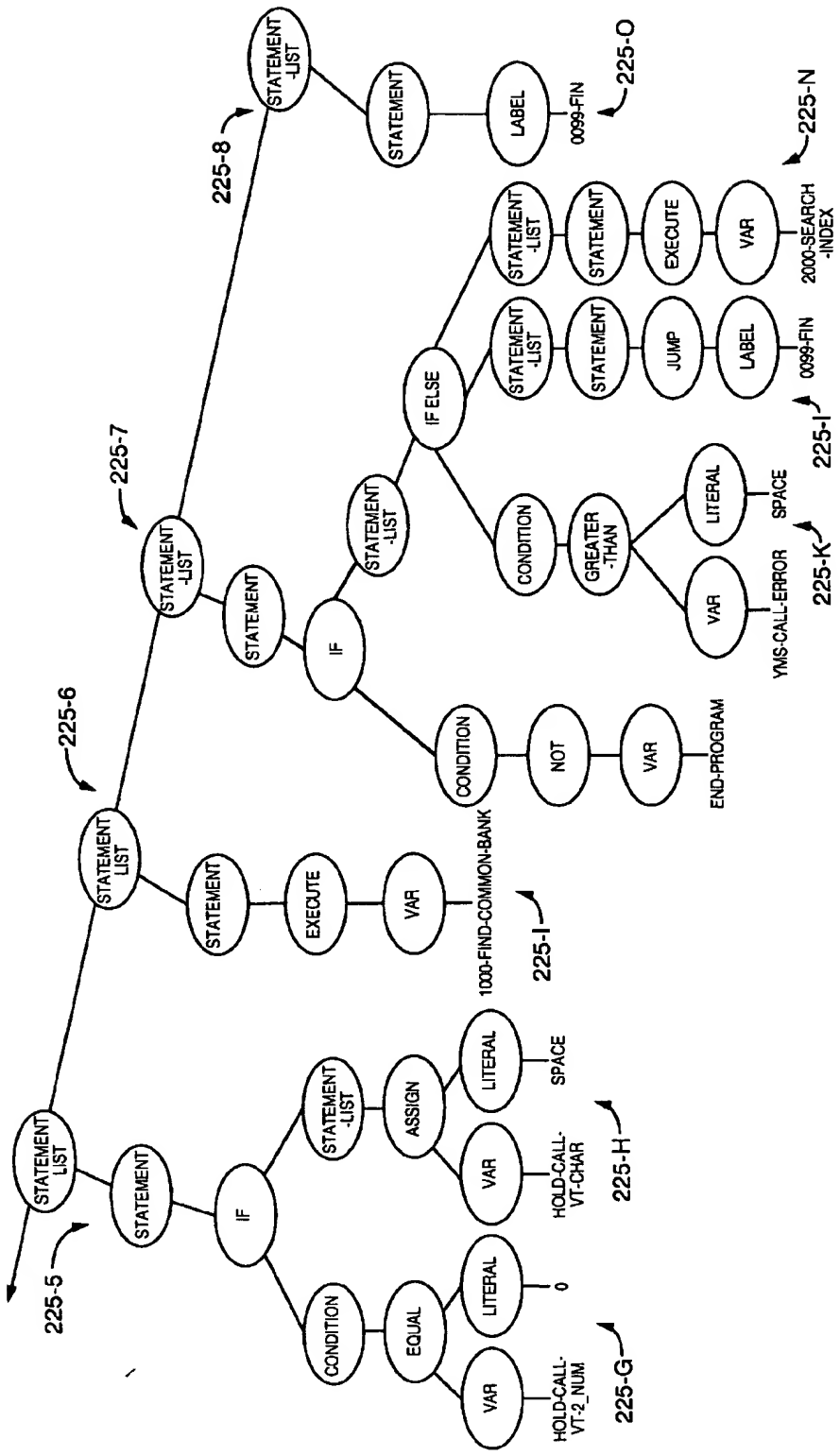

STATEMENT LIST
225-5
STATEMENT
IF
CONDITION
EQUAL
VAR
HOLD-CALL-VT-2_NUM
225-G
LITERAL
0
STATEMENT-LIST
ASSIGN
VAR
HOLD-CALL-VT-CHAR
225-H
LITERAL
SPACE
225-6
STATEMENT LIST
STATEMENT
EXECUTE
VAR
1000-FIND-COMMON-BANK
225-I
225-7
STATEMENT-LIST
STATEMENT
IF
CONDITION
NOT
VAR
END-PROGRAM
STATEMENT-LIST
IF ELSE
CONDITION
GREATER-THAN
VAR
YMS-CALL-ERROR
LITERAL
SPACE
225-K
STATEMENT-LIST
STATEMENT
JUMP
LABEL
0099-FIN
225-I
225-8
STATEMENT-LIST
STATEMENT
LABEL
0099-FIN
225-O
STATEMENT-LIST
STATEMENT
EXECUTE
VAR
2000-SEARCH-INDEX
225-N

FIG. 20B

VMS-CALL-RESULT VAR FLAG-AREA VAR VARLIST LOW-VALUE VAR ASSIGN STATEMENT 225-1'

YMS-CALL-AREA VAR 9000 LITERAL GREATER-THAN CONDITION YMS-CALL-ERROR VAR N LITERAL ASSIGN STATEMENT 0099-FIN LABEL JUMP STATEMENT STATEMENT-LIST IF STATEMENT 225-2'

BIT-NUM VAR 0 LITERAL ASSIGN STATEMENT 225-3'

HOLD-CALL-VT VAR YMS-CALL-VT VAR ASSIGN STATEMENT 225-4'

HOLD-CALL-VT-2-NUM VAR 0 LITERAL EQUAL CONDITION HOLD-CALL-VT-CHAR VAR SPACE LITERAL ASSIGN STATEMENT-LIST IF STATEMENT 225-5'

1000-FIND-COMMON-BANK VAR EXECUTE STATEMENT 225-6'

END-PROGRAM VAR NOT CONDITION YMS-CALL-ERROR VAR SPACE LITERAL GREATER-THAN CONDITION 0099-FIN LABEL JUMP STATEMENT STATEMENT-LIST 2000-SEARCH-INDEX VAR EXECUTE STATEMENT STATEMENT-LIST IF-ELSE STATEMENT-LIST IF STATEMENT 225-7'

0099-FIN LABEL STATEMENT 225-8'

STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST STATEMENT-LIST 225-9'

225'

FIG. 21

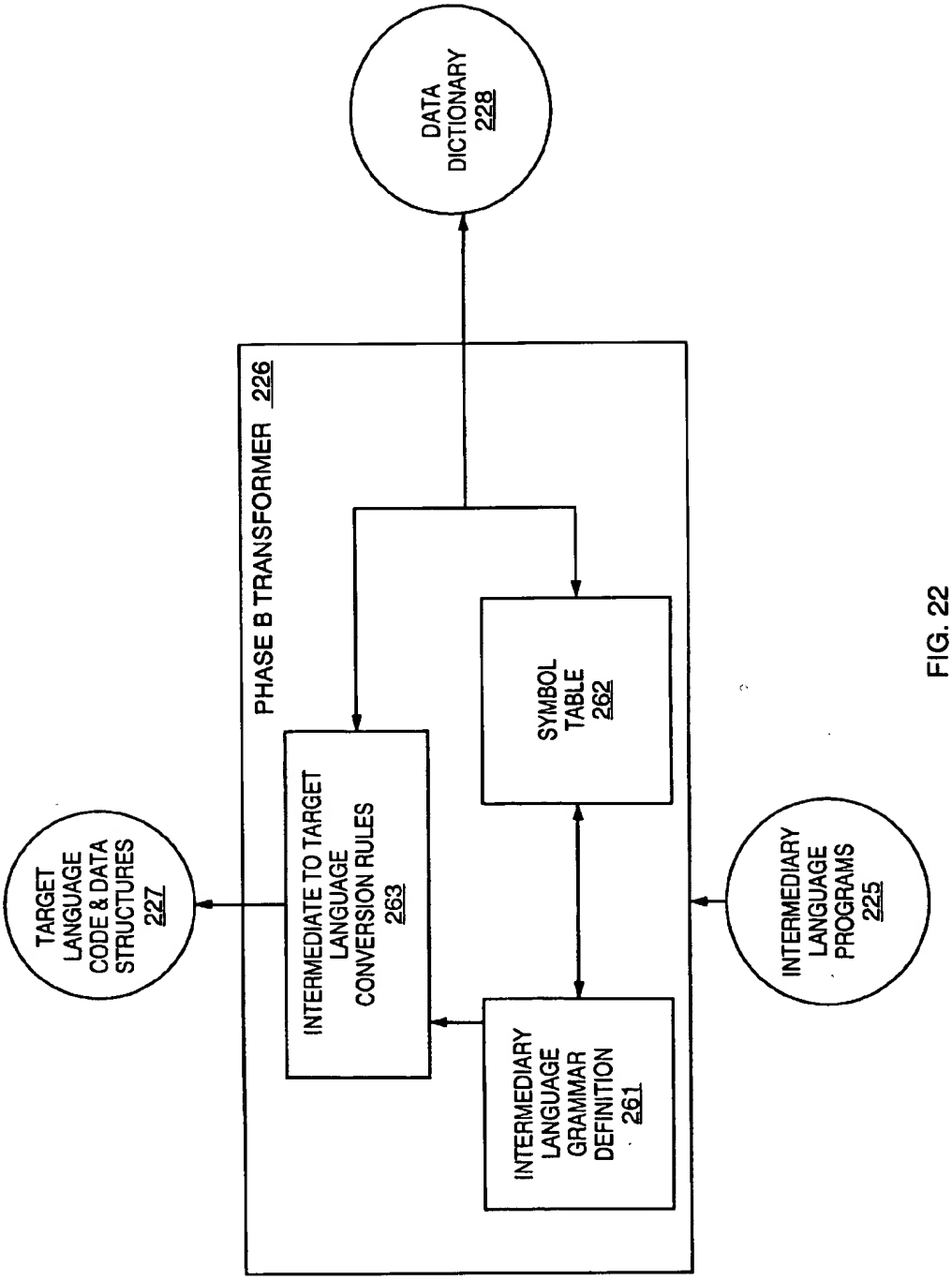

DATA DICTIONARY 228
PHASE B TRANSFORMER 226
SYMBOL TABLE 262
INTERMEDIATE TO TARGET LANGUAGE CONVERSION RULES 263
INTERMEDIARY LANGUAGE GRAMMAR DEFINITION 261
TARGET LANGUAGE CODE & DATA STRUCTURES 227
INTERMEDIARY LANGUAGE PROGRAMS 225

FIG. 22

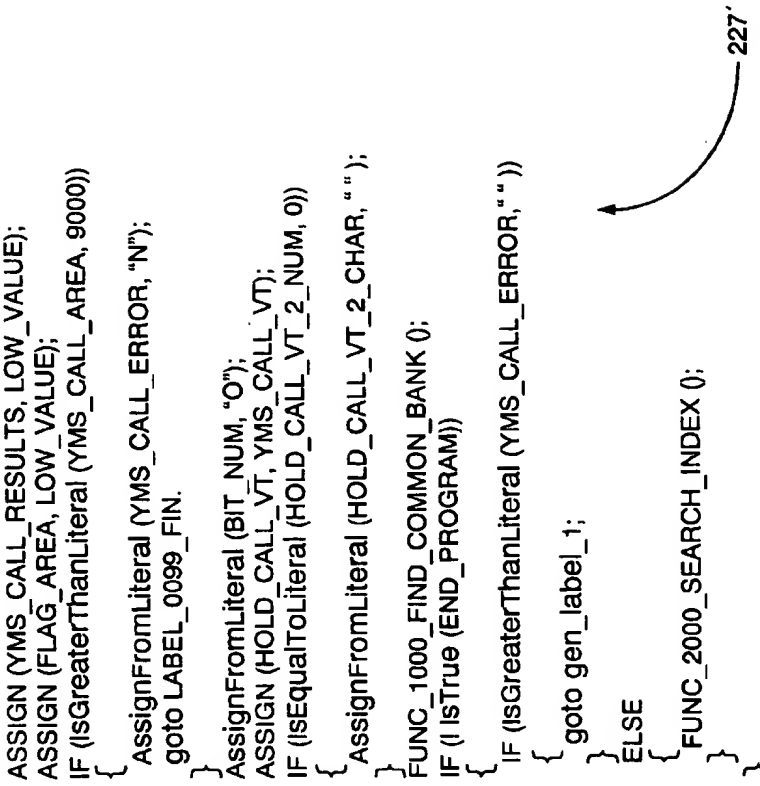

ASSIGN (YMS_CALL_RESULTS, LOW_VALUE);
ASSIGN (FLAG_AREA, LOW_VALUE);
IF (IsGreaterThanLiteral (YMS_CALL_AREA, 9000))
{
AssignFromLiteral (YMS_CALL_ERROR, "N");
goto LABEL_0099_FIN.
}
AssignFromLiteral (BIT_NUM, "0");
ASSIGN (HOLD_CALL_VT, YMS_CALL_VT);
IF (IsEqualToLiteral (HOLD_CALL_VT_2_NUM, 0))
{
AssignFromLiteral (HOLD_CALL_VT_2_CHAR, " ");
}
FUNC_1000_FIND_COMMON_BANK ();
IF (! IsTrue (END_PROGRAM))
{
IF (IsGreaterThanLiteral (YMS_CALL_ERROR, " "))
{
goto gen_label_1;
}
ELSE
{
FUNC_2000_SEARCH_INDEX ();
}
}
gen_label_1: /* NEXT SENTENCE */
LABEL_0099_FIN:
227'

FIG. 23A

```
YMS_CALL_RESULTS = FLAG_AREA = LOW_VALUE;

IF (YMS_CALL_AREA > 9000 )
{
 YMS_CALL_ERROR = "N";
 goto LABEL_0099_FIN;
}

BIT_NUM = 0;
HOLD_CALL_VT = YMS_CALL_VT;

IF (HOLD_CALL_VT_2_NUM ==0 )
{
 HOLD_CALL_VT_2_CHAR = " ";
}

FUNC_1000_FIND_COMMON_BANK ();

IF (! END_PROGRAM)
{
 IF (( YMS_CALL_ERROR > " ")
 {
  goto gen_label_1;
 }
 ELSE
 {
  FUNC_2000_SEARCH_INDEX ();
 }
}

gen_label_1: /* NEXT SENTENCE */
LABEL_0099_FIN
```

227'

FIG. 23B

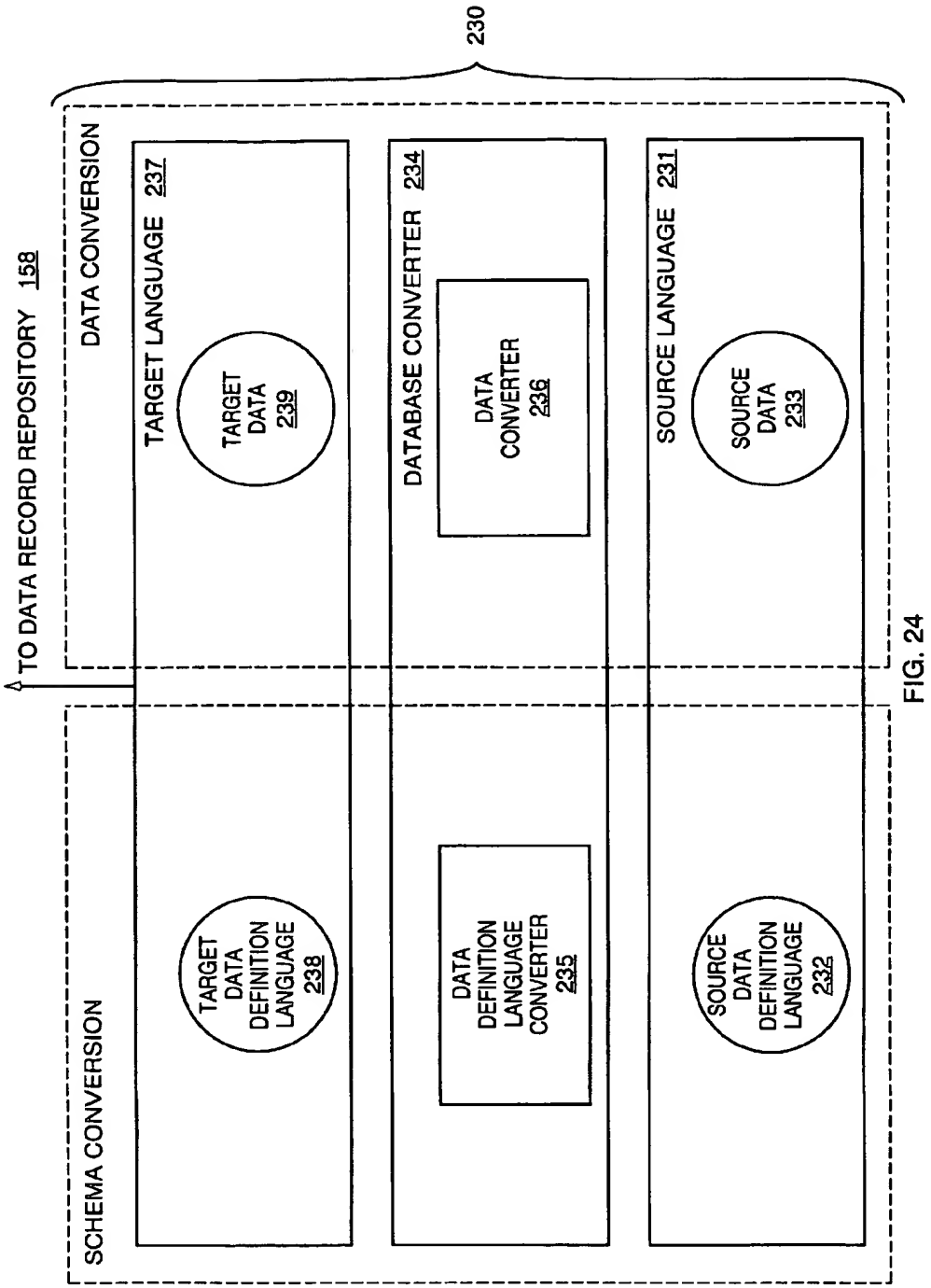

230
DATA CONVERSION
TARGET LANGUAGE 237
TARGET DATA 239
DATABASE CONVERTER 234
DATA CONVERTER 236
SOURCE LANGUAGE 231
SOURCE DATA 233
TO DATA RECORD REPOSITORY 158
SCHEMA CONVERSION
TARGET DATA DEFINITION LANGUAGE 238
DATA DEFINITION LANGUAGE CONVERTER 235
SOURCE DATA DEFINITION LANGUAGE 232

FIG. 24

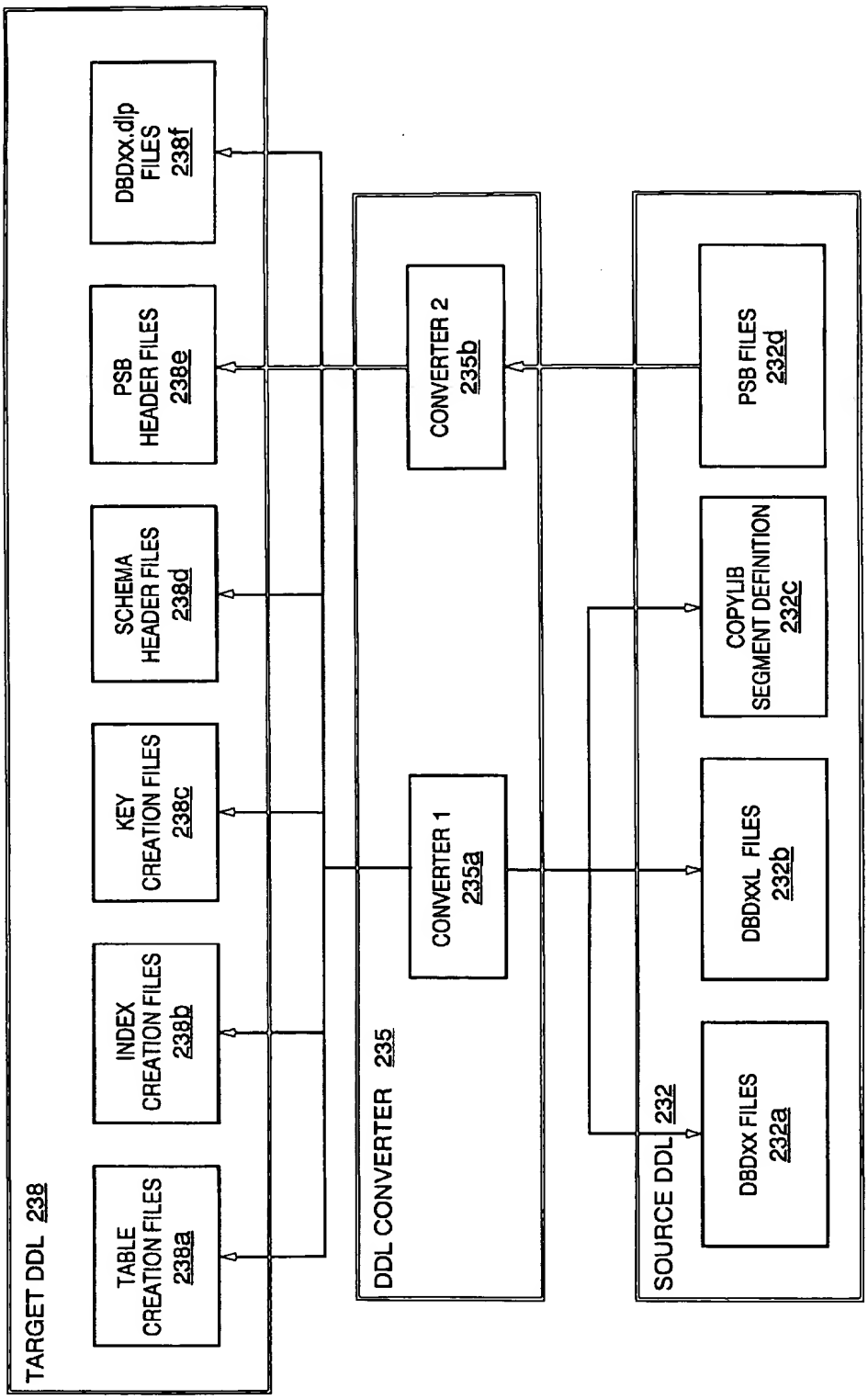

TARGET DDL 238
TABLE CREATION FILES 238a
INDEX CREATION FILES 238b
KEY CREATION FILES 238c
SCHEMA HEADER FILES 238d
PSB HEADER FILES 238e
DBDxx.dlp FILES 238f
DDL CONVERTER 235
CONVERTER 1 235a
CONVERTER 2 235b
SOURCE DDL 232
DBDxx FILES 232a
DBDxxL FILES 232b
COPYLIB SEGMENT DEFINITION 232c
PSB FILES 232d

FIG. 25

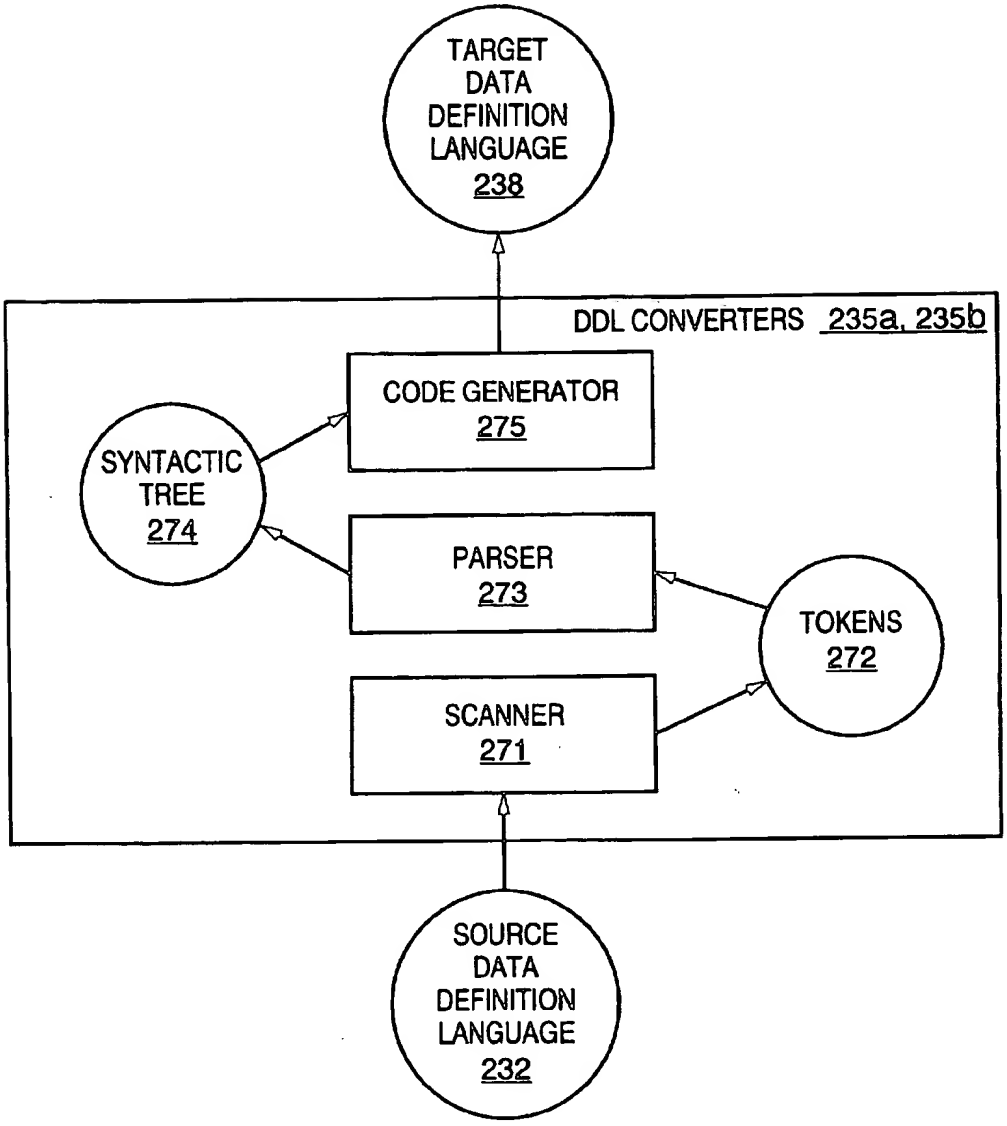

TARGET DATA DEFINITION LANGUAGE 238
DDL CONVERTERS 235a, 235b
CODE GENERATOR 275
SYNTACTIC TREE 274
PARSER 273
TOKENS 272
SCANNER 271
SOURCE DATA DEFINITION LANGUAGE 232

FIG. 26

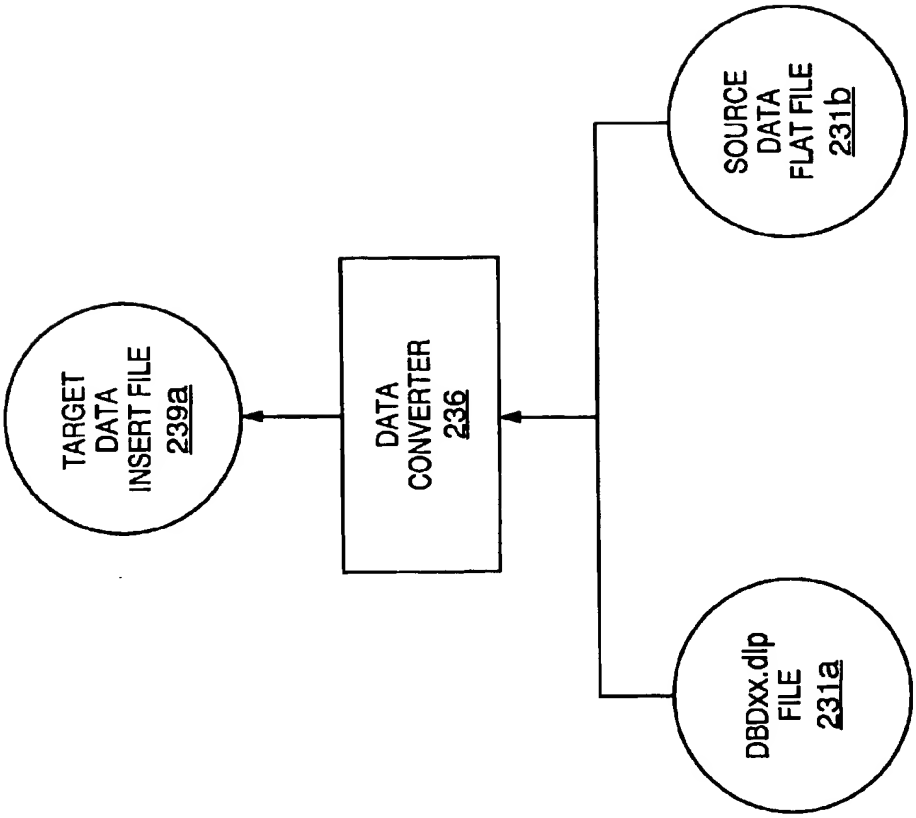

SOURCE DATA FLAT FILE 231b
DATA CONVERTER 236
TARGET DATA INSERT FILE 239a
DBDxx.dlp FILE 231a

FIG. 27

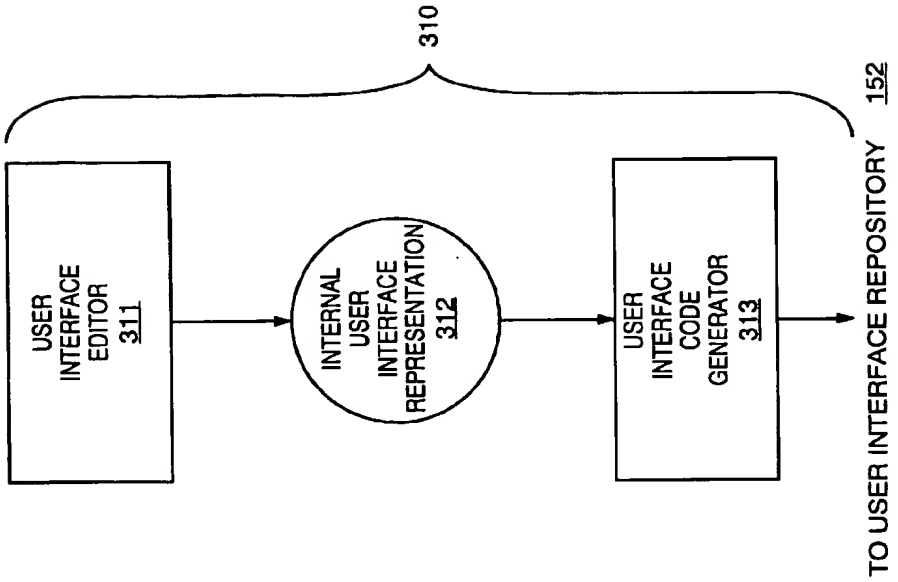

310
USER INTERFACE EDITOR 311
INTERNAL USER INTERFACE REPRESENTATION 312
USER INTERFACE CODE GENERATOR 313
TO USER INTERFACE REPOSITORY 152

FIG. 28

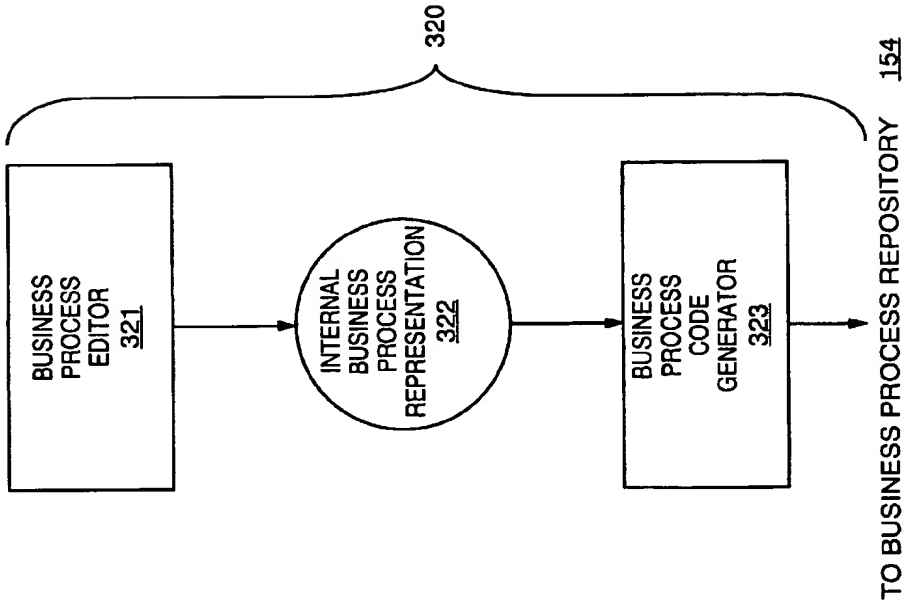

320
BUSINESS PROCESS EDITOR 321
INTERNAL BUSINESS PROCESS REPRESENTATION 322
BUSINESS PROCESS CODE GENERATOR 323
TO BUSINESS PROCESS REPOSITORY 154

FIG. 29

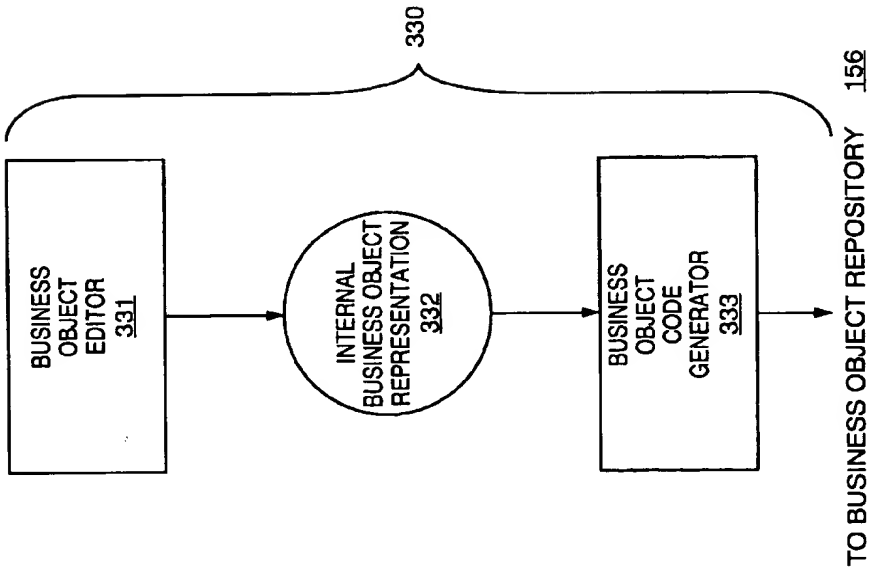

330
BUSINESS OBJECT EDITOR 331
INTERNAL BUSINESS OBJECT REPRESENTATION 332
BUSINESS OBJECT CODE GENERATOR 333
TO BUSINESS OBJECT REPOSITORY 156

FIG. 30

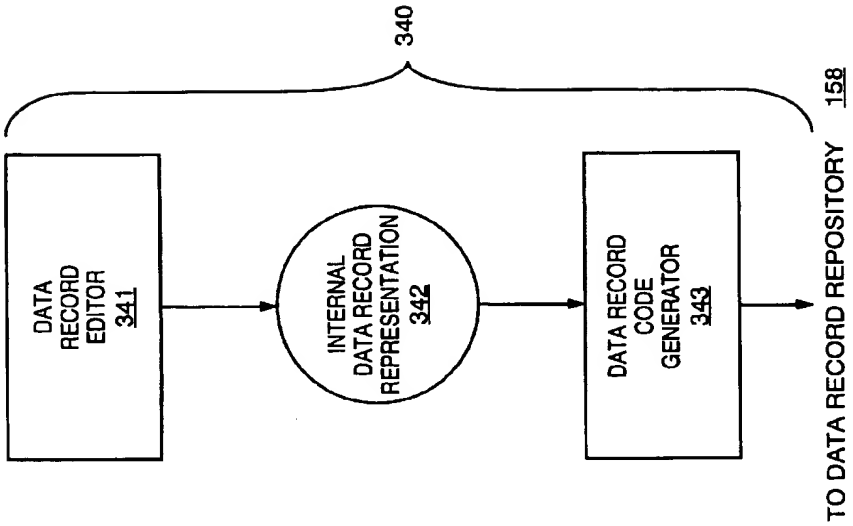

340
DATA RECORD EDITOR 341
INTERNAL DATA RECORD REPRESENTATION 342
DATA RECORD CODE GENERATOR 343
TO DATA RECORD REPOSITORY 158

FIG. 31

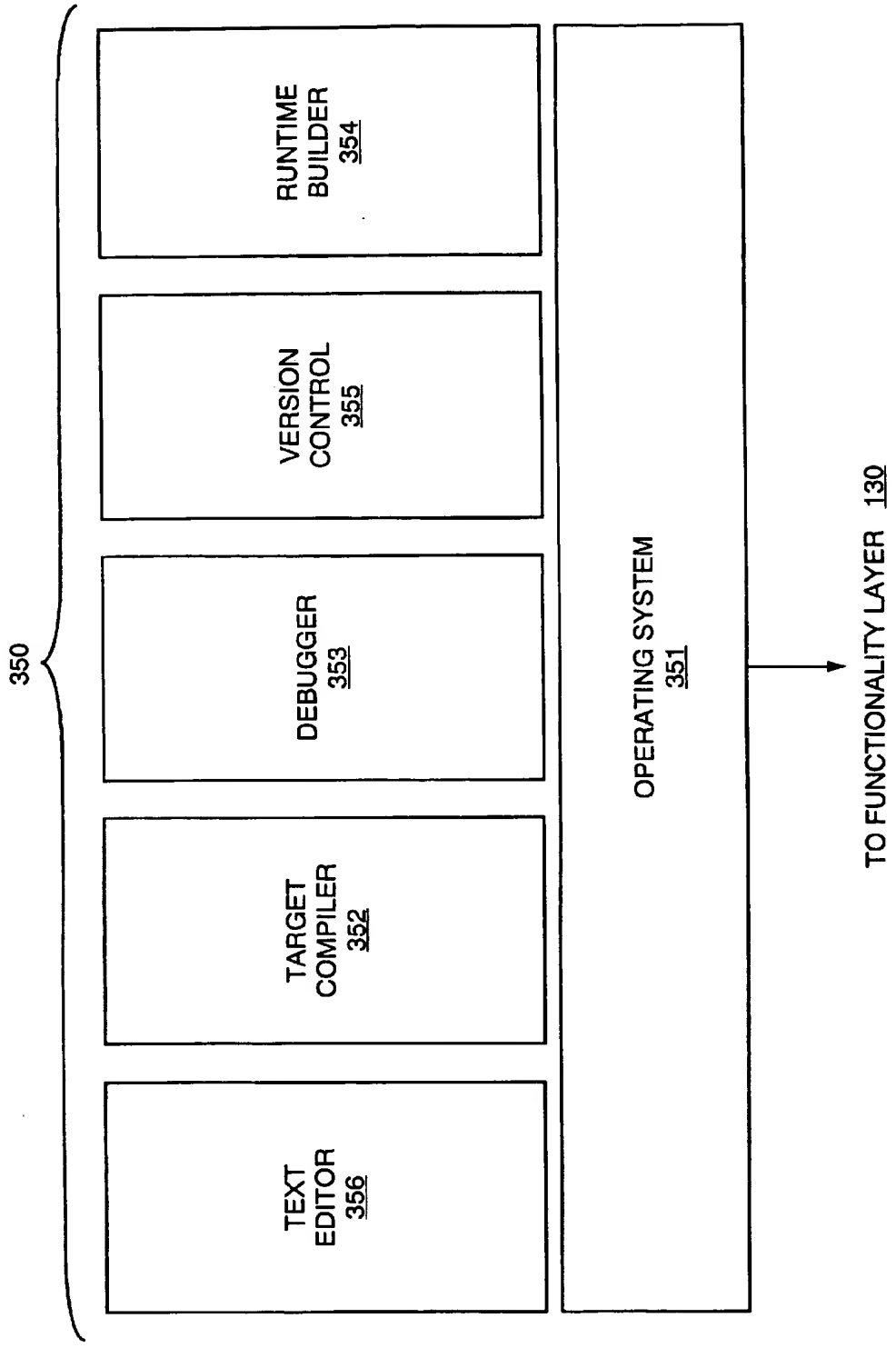

350
TEXT EDITOR 356
TARGET COMPILER 352
DEBUGGER 353
VERSION CONTROL 355
RUNTIME BUILDER 354
OPERATING SYSTEM 351
TO FUNCTIONALITY LAYER 130

FIG. 32

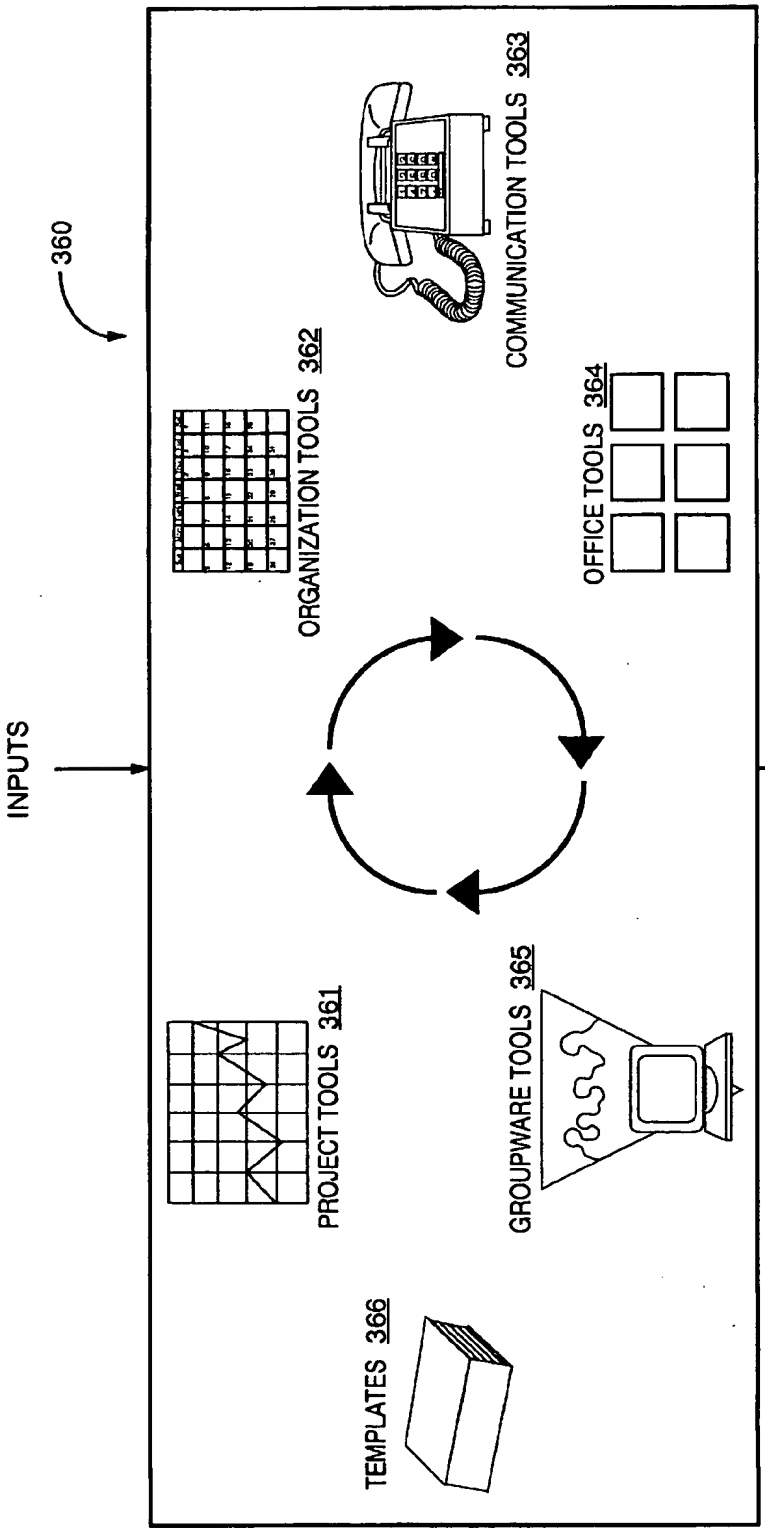

360
INPUTS
COMMUNICATION TOOLS 363
ORGANIZATION TOOLS 362
OFFICE TOOLS 364
PROJECT TOOLS 361
GROUPWARE TOOLS 365
TEMPLATES 366
OUTPUTS

FIG. 33

SYSTEM TO TRANSITION AN ENTERPRISE TO A DISTRIBUTED INFRASTRUCTURE

RELATED APPLICATIONS

This application claims priority to U.S. Provisional Application No. 60/016,330 filed on May 3, 1996, the teachings of which are incorporated herein by reference in their entirety.

BACKGROUND

Corporations characteristically use applications executing on computer systems to automate their business functions. The applications typically contain parts that deal with the user interface, parts that deal with business processes, parts that deal with programming logic, and parts that deal with data. The applications are typically built to operate on a single computer platform. In this context, a computer platform includes the programs, or software, to perform the various tasks required of a computer, as well as the machine, or hardware, that hosts these programs.

Given the large amount of business functions that need to be automated within a corporation, applications are often centralized on one single platform. From a hardware point of view, such platforms include large and powerful computers such as mini-computers or mainframes. On the software side, such platforms often take the form of all-encompassing environments that meet all of the application development needs, such as Information Management System/Virtual Storage (IMS/VS) from International Business Machines Corporation (IBM). In the IMS/VS model, the various conceptual layers constituting an application are bundled together in one single piece or a small number of interdependent pieces. This single-tier model is well understood, reliable, and secure. It facilitates control and limits overhead.

The proprietary nature of the host platform, however, leads to severe economic disadvantages. Initial platform costs are sizable, and subsequent growth is limited by the capacity of the hardware and software components of the platform that hosts the application. Furthermore, application maintenance and enhancement is a complex and cumbersome process. Application users are removed from application developers, increasing the gap between requirements and implementation. Changes to one part of the application also require compatible changes to the other parts of the application. Finally, data access and communication standards are limited to those supported by the host platform.

The advent of desktop computing in the form of the personal computer provides an initial solution to the above problems by enabling corporations to depart from the centralized platform model. To maximize the usage of their computer resources, applications can be distributed among different platforms. This distributed system approach can take many forms, the most widespread of which is known as the two-tiered client/server architecture. This two-tiered model divides the application between two platforms: a client and a server. At a high level, clients and servers are software concepts. A client makes requests from servers, while servers provide adequate services to fulfill client requests. Client hosts are typically personal computers, while server hosts are typically mini-computers or mainframe computers.

In the client/server model, the presentation part of the application is usually located on the client platform and the data part of the application is found on the server platform, with the business process and functionality parts of the application merged into either of the other two layers. This model is more economical than the single-tier model, with less or no hardware lock-in. The division into two tiers is also more flexible to user interface changes. On the other hand, the proprietary nature of the platform is still present at the level of the software.

SUMMARY OF THE INVENTION

Even in the two-tiered client/server model, the mixing of functionality with either presentation or data still leads to complicated code changes. Also, data access and communication standards are still limited. Furthermore, enterprises have no facilities to transition their existing infrastructure to a distributed computing model.

A preferred embodiment of the invention includes a system to transition an entire business enterprise to a distributed infrastructure. The distributed infrastructure is preferably a multi-tiered client/server target architecture that adheres to open system standards. The multi-tiered architecture preferably includes at least four layers including a separate process control layer and functionality layer. The process control layer includes a state router to control work flow in accordance with the business procedures of the enterprise. The functionality layer includes modules for performing the work. The architecture also preferably includes a presentation layer for interfacing with a user, and a data retrieval layer for accessing data stored in a separate data storage layer.

A transition of an entire business enterprise to a distributed infrastructure based on the new architecture is performed using a process for organizing and managing the transition. Notably, this requires that each legacy (source) application be identified and prioritized. For each source application, there are a range of available transition choices, including the option of translating the source application to the new target architecture without changing any of the existing functionality and the option of re-engineering the source application by changing the existing functionality. The source application may also be replaceable by a commercial product or a custom application written in-house. The source applications are then transitioned in order of priority to the new architecture.

Specifically, a preferred system in accordance with the present invention includes the automated capability to translate existing source applications into new target applications on a multi-tiered client/server architecture. The translation of source applications to target applications includes the conversion of user interfaces, procedural languages, and data definitions. These conversions use a two-phase process where source program components written in the source languages are first translated to components in a common intermediate language. The intermediate language components are then translated to target program components in the target languages. By using a common intermediate language, only one translation module is required for each source and target language.

A preferred system in accordance with the present invention further includes a facility to create a new application based on the multi-tiered client/server architecture and to modify an existing application that already uses this multi-tiered client/server architecture.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the invention, including various novel details of construction and combination of parts will be apparent from the

following more particular drawings and description of preferred embodiments of a system to transition an enterprise to a distributed infrastructure in which the reference characters refer to the same parts throughout the different views. It will be understood that the particular apparatus and methods embodying the invention are shown by way of illustration only and not as a limitation of the invention, emphasis instead being placed upon illustrating the principles of the invention. The principles and features of this invention may be employed in various and numerous embodiments without departing from the scope of the invention.

FIG. **1** is a high level block diagram of a system embodying the invention.

FIG. **2** is a functional block diagram of the interrelationships of FIG. **1**.

FIG. **3** is a schematic block diagram of a process for managing the transition of an entire enterprise to a distributed infrastructure.

FIG. **4** is a schematic diagram of the presentation layer **110** of FIG. **1**.

FIG. **5** is a schematic diagram of a sample mapping between application user interface representation structures **116** and display platform user interface representation structures **118**.

FIG. **6** is a block diagram of the operational modules of the user interface engine **117** of FIG. **4**.

FIG. **7** is a schematic diagram illustrating the business process layer **120**, functionality layer **130**, and data access layer **140** of FIG. **1**.

FIG. **8** is a block diagram of the business process layer **120** of FIG. **1**.

FIG. **9** is a flow diagram of the communication mechanism between the user interface engine **117** and the state router **122**.

FIG. **10** is a flow diagram of the operations of a particular preferred state router **122** of FIG. **8**.

FIG. **11** is a block diagram of the functionality layer **130** of FIG. **1**.

FIG. **12** is a block diagram of the data access layer **140** of FIG. **1**.

FIG. **13** is a block diagram of the data storage layer **150** of FIG. **1**.

FIG. **14** is a schematic diagram of a hardware platform for the data storage layer **150** of FIG. **19**.

FIG. **15** is a block diagram of the control layer **160** of FIG. **1**.

FIG. **16** is a block diagram of the user interface conversion utility **210** of FIG. **1**.

FIG. **17** is a flow diagram of the procedural language conversion utility **220** of FIG. **1**.

FIG. **18** is an exemplary source code fragment.

FIG. **19** is a block diagram of the first phase transformer program **224** of FIG. **17**.

FIG. **20** illustrates in FIGS. **20A–20B** a parse tree for the source code fragment of FIG. **18**.

FIG. **21** is an intermediate language file for the source code fragment of FIG. **18**.

FIG. **22** is a block diagram of the second phase transformer program **22** of FIG. **17**.

FIGS. **23A–23B** are C and C++ target code fragments there being source code fragment of FIG. **18**.

FIG. **24** is a block diagram of the data definition language conversion utility **230** of FIG. **1**.

FIG. **25** is a block diagram illustrating a schema conversion.

FIG. **26** is a block diagram of a converter **235***a*, **235***b* of FIG. **25**.

FIG. **27** is a block diagram of the data converter **236** of FIG. **24**.

FIG. **28** is a block diagram of the graphical user interface editor **310** of FIG. **1**.

FIG. **29** is a block diagram of the graphical business process editor **320** of FIG. **1**.

FIG. **30** is a block diagram of the graphical business object editor **330** of FIG. **1**.

FIG. **31** is a block diagram of the graphical data record editor **340** of FIG. **1**.

FIG. **32** is a block diagram of the logic development environment **350** of FIG. **1**.

FIG. **33** is a schematic block diagram of the facilitation tools **360** of FIG. **1**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. **1** is a high level block diagram of a system embodying the invention. As shown, the system **1** includes a multi-tiered architecture **10**, a re-architecting system **20** for converting source applications of an enterprise to target applications on the multi-tiered architecture **10**, and a re-engineering subsystem **30** for custom application development or re-engineering.

A preferred multi-tiered architecture **10** of the present invention comprises at least four separate layers. As illustrated, the architecture **10** includes at least one presentation layer **110**, one separate business process layer **120**, one separate functionality layer **130**, one separate data access layer **140**, one separate data storage layer **150**, and one separate control layer **160**, all inter-connected through communication links **170**. The data storage layer **150** preferably includes a user interface repository **152**, a business process repository **154**, a business object repository **156**, and a data record repository **158** for storing data.

A preferred re-architecting system **20** includes a user interface conversion utility **210**, a procedural language conversion utility **220**, and a data definition language conversion utility **230**. The procedural language conversion utility **220** is in communication with the functionality layer **130** and the data access layer **140** of the multi-tier architecture **10**. The user interface conversion utility **210** is in communication with the user interface repository **152** and the data definition language conversion utility **230** is in communication with the data record repository **158**.

A preferred re-engineering system includes a graphical user interface editor **310**, a graphical business process editor **320**, a graphical business object editor **330**, a graphical data editor **340**, a logic development environment **350**, and facilitation tools **360**. The logic development environment **350** is in communication with the functionality layer **130** of the multi-tier architecture **10**. The graphical user interface editor **310**, the graphical business process editor **320**, the graphical business object editor **330**, and the graphical data record editor **340** are in communication with the user interface repository **152**, the business process repository **154**, the business object repository **156**, and the data record repository **158**, respectively.

Even though the re-engineering system **30** is an integral part of the overall system of the present invention, it is not part of the actual transition of an enterprise to a distributed

infrastructure. Instead, the re-engineering system **30** enables the enterprise to maintain and enhance its distributed infrastructure once the transition itself is complete.

In its simplest form, a host for all the layers can be a single platform. At the other end of the spectrum, each layer can be hosted on a different platform. In the spirit of distributed systems, a preferred embodiment of the present invention hosts each layer on a separate platform, both in terms of hardware and software. In a preferred embodiment of the present invention, the architecture **10** supports both custom-developed applications as well as application converted from a legacy system. The IMS/VS legacy environment is used herein to illustrate, but not limit, architectural concepts that pertain to a converted legacy application.

FIG. **2** is a functional block diagram of the interrelationships of FIG. **1**. Conceptually, the user interface translator **210** and the graphic user interface editor **310** affect the presentation layer **110** of the multi-tiered architecture **10**. The graphical business process editor **320** affects the business process layer **120**. The procedural language conversion utility **220**, the graphical business object editor **330**, and a logic development environment **350** conceptually affect the functionality layer **130**. The procedural language conversion utility **220** also conceptually affects the data access layer **140**. The data definition language translator **230** and the graphical data record editor **340** conceptually affect the data storage layer **150**.

FIG. **3** is a schematic block diagram of a preferred process for managing the transition of an entire enterprise to a distributed infrastructure. Preferably, the transition management process **40** includes a series of high level serial stages, including an implementation strategy stage **42**, an implementation planning stage **44**, a system implementation stage **46**, and an operation stage **48**. As the transition management process **40** proceeds from the implementation strategy stage **42** through to the operation stage **48**, the amount of operations support required by the legacy system decreases and the amount of operations support for the open system increases, as illustrated.

The implementation strategy stage **42** is a sequence of manual steps embodied in a Business Case and Implementation Strategy (BCIS) process **420**. The BCIS process **420** determines the business and technology drivers of the enterprise, builds a business case analysis of the economic feasibility of the transition, and creates an implementation strategy that provides the basic outline of the organization of the transition. This can involve studies of the existing and envisioned organization, infrastructure, and technology, as well as an evaluation of the impact of the transition in these areas. A study of the organization analyzes methods for developing the skills necessary for the transition and to manage potential resistance to change. A study of the infrastructure analyzes the hardware, software, and network required to support the transition. A technology study analyzes the tools, architectures, and other technologies necessary for the transition. Although the implementation strategy stage **42** is described as a manual process, an expert system can be employed to perform some or all of the processing.

More specifically, the focus is first to define the current state of the enterprise, which provides the start point. This start point is defined through an assessment of the performance of the existing information technology in supporting current business goals, an assessment of the readiness of the organization to build and support a new environment, and an assessment of the limitations inherent to the existing infrastructure.

The next focus is to identify the mission of the enterprise. The mission will give a good idea of the on-going direction of the enterprise, with which all undertakings must be aligned. Once the mission is identified, a conceptual vision for the future of the enterprise is defined, consistently with the mission.

Unlike the mission, the conceptual vision is not continuous, but must be attained within a predetermined time frame. This conceptual vision provides the destination point. Given its conceptual nature, the corporate vision must now be broken down into specific goals or objectives, that are numbered for reference. Outputs are then associated with each objective, as measurable outcomes that will clearly indicate the achievement of the associated objective. To secure this association, outputs are numbered consistently with their corresponding objectives.

Now that a start point, a direction, and a destination point have been defined, the intermediate steps that lead to each outcome must be delineated. These constitute the factors that are critical to successful achievement of these outputs, and are consequently referred to as the Critical Success Factors (CSF). Each output is given a set of CSFs associated with it, with appropriate numbering for reference. At this point, a strategy is put into place for achieving all of the CSFs for each of the outputs. Each strategy is numbered consistently with the output at which it is meant to arrive. Finally, a set of specific short term action items is associated with each strategy, as a means to move from the current situation towards the first CSF for each output. As before, action items are numbered consistently with the CSFs they are meant to lead towards. The implementation strategy stage **42** provides a start point for the implementation planning stage **44**. The implementation planning stage **44** includes an Applications and Process Portfolio Analysis (APPA) process **442**, followed by a series of Applications Staging and Planning (ASAP) **444** sessions. The APPA process and the ASAP sessions can be performed manually or with the aid of expert systems.

The intent of the APPA process **442** is to gather and document an inventory of all the current and envisioned applications and business processes of an enterprise. The APPA process **442** separates the strategic from the tactical, and for each one, determines the transition that needs to take place to move from the existing to the envisioned situation.

The range of transition possibilities include: do nothing, re-architect, re-engineer, re-architect and then re-engineer, replace by an off-the-shelf commercial solution, replace by a custom solution built in-house, and integrate. The do nothing option retains the existing application or process as is. The re-architect option translates the existing application to the new architecture without changing any of the existing functionality. The re-engineer option changes the existing functionality while remaining on the same architecture. The option of combining re-architecting and re-engineering first translates the existing application into the new architecture, and then modifies the application functionality in the context of the new architecture. The option of replacing by an off-the-shelf commercial solution replaces the existing application with a corresponding application package which is used, with or without customization, to perform the function of the replaced application. This solution is not an alternative for strategic applications and processes due to the inherent strategic nature of such solutions and to the loss of competitive advantage of the enterprise should these solution be built using tools and methods commercially available and thus easily reproducible. The option of replacing by a custom solution built in-house replaces the existing appli-

cation with a new application built from scratch, using the new architecture. The integration option combines the various applications (whether already present, re-architected, re-engineered, purchased, or custom developed) into the new architecture to obtain a coherent infrastructure based on the new architecture.

Once this inventory is completed, applications and processes are prioritized, and the planning of the transition can be initiated, starting with the applications and processes with the highest priority. This transition planning corresponds to the ASAP process **444**, which focuses on a single application or process. The ASAP process **444** focuses on the details of the existing and envisioned application or process, evaluates the scope of the transition effort, and prepares a detailed implementation plan to conduct the transition, including tasks, schedule, and resources. An ASAP process **444** is thus carried out for each application and process identified during the APPA process **442**, in order of decreasing priority.

Once the implementation planning stage **44** is completed as described above, the actual implementation stage **46** can be initiated. Depending on the transition alternative selected for a particular application or process, a different implementation process may be applied. At the implementation stage **46**, multiple applications and processes can go through the transition in parallel.

A Strategic Application Advancement (SAA) process **461** is an automated implementation process that focuses on re-architecting. Re-architecting preserves an application's core functionality intact, transformed into a multi-tiered client/server architecture. Re-architecting is often followed by re-engineering to add or change existing functionality to accommodate new business processes. Re-architecting involves identifying the business goals, objectives, and processes encompassed by the system, determining the existing source and desired target architectures, defining information systems standards, determining infrastructure requirements, performing the actual conversion, providing any re-engineering required, including design documentation for re-engineering requirements and technical documentation for re-architected application maintenance, and empowerment of staff for application maintenance and enhancements.

A Strategic Applications System Development (SASD) process **463** is a manual implementation process that focuses on re-engineering or custom development. The SASD process **463** encompasses the design and development of re-engineered or custom-built multi-tiered client/server applications conforming to open system industry standards. Re-engineering refers to modifications to an existing system, usually the product of a prior re-architecting effort. Re-engineering must take into account the maintainability and performance issues that arise when attempting substantial changes to an application designed for a legacy system and converted to a multi-tiered client/server architecture. Custom development refers to the creation of a multi-tiered client/server application from user requirements. Consequently, custom development follows familiar application life-cycle steps.

A Tactical Applications Planning and Implementation (TAPI) process **467** is a manual implementation process that focuses on the usage of commercial off-the-shelf packages. This involves selection and customization of commercial packages to achieve reusable applications in very short time frames. The TAPI process **467** is targeted to tactical as opposed to strategic applications, because such applications are not critical to the competitive posture of the business and therefore can make use of available packaged technology without endangering the competitiveness of an enterprise.

An Open Systems Integration (OSI) process **469** is a manual implementation process that focuses on integrating applications that are purchased, newly custom developed, re-architected, or re-engineered to share data and screens. This process includes the definition of business goals and objectives, the definition of applicable business processes, the study of application interactions and data relationships, and the planning of hardware and software infrastructures. The OSI process **469** also includes the implementation of the integration, including detailed implementation plan and schedule and detailed requirements and design documentation, user acceptance testing, comprehensive technical documentation, and empowerment of support staff for the maintenance phase. One powerful example of integration at the user interface layer using the OSI process **469** is the creation of a corporate intranet using internet HyperText Manipulation Language (HTML) or a highly-level language generating HTML, such as Java from Sun Microsystems to provide a user-friendly, platform independent, common user interface to corporate application.

Implementation also includes a Skills Enhancement and Empowerment (SEE) process **462** and an Operations Process Advancement (OPA) process **468**. The SEE process **462** focuses on organizational issues during implementation, such as changing management and personnel training. The OPA process **468** focuses on operational support of the applications developed by the other implementation processes, including standardization of tools and processes, infrastructure setup, and system operation.

Once the implementation stage **46** is completed, the operations stage **48** begins. The operations stage **48** includes a Customer Service and Support (CSS) process **480**, which can include initial or continued application maintenance and user support, full-size production and operations, and possibly outsourcing of all information system needs. As mentioned previously, the facilitation tools of the re-engineering system are available for electronic planning, tracking, and documentation of all the stages of the transition management process **40**.

FIG. **4** is a schematic diagram of the presentation layer **110** of FIG. **1**. In its simplest form, the presentation layer **110** can be implemented using a conventional personal computer. It can also take the form of an X-terminal, a workstation console, or a Macintosh style interface display. As shown, the presentation layer **110** includes a processor **111** having the current screen representation, constructed according to the principles of the present invention. The processor is preferably a user workstation which includes a display unit through which commands or user selections can be entered via a keyboard or mouse. The processor **111** also includes internal or external storage, such as a disk device, from which a user interface engine is loaded into the memory of the processor **111** as required. For a personal computer, X-terminal, or workstation having a large main memory storage, the entire application front-end can remain resident, thereby enhancing system performance. The storage unit is also used to store presentation layer log files. The presentation layer **110** can further include a printer **113**, connected to the processor **111** through a communication link, such as a parallel or serial port. The printer **113** can be used to provide a permanent record of application log files, reports, source code, or screen listings according to the present invention.

As shown in FIG. **4**, the presentation layer **110** includes a user interface display platform **115**, an application user

interface representation mechanism **116**, and a user interface engine **117**. In a preferred embodiment of the present invention, the user interface display platform **115** is a conventional Graphical User Interface (GUI) tool, commercially available. Consequently, the user interface display platform **115** has its own internal user interface representation mechanism **118** to display the various components of a user interface, usually in a graphical way.

Preferably, the underlying internal user interface of the user interface display platforms **115** is preferably derived from a frame-based system. A frame system is a network of frames and relations, corresponding to the nodes and links of a mathematical graph. Frame systems are organized in a hierarchy in which the high-level frames represent more general concepts and the lower frames represent more specific concepts. At the lowest levels, the frames represent instances of those concepts. The concept at each frame is defined by a collection of attributes or properties which can have values and, in this respect, the frames and attributes in a frame system are comparable to the records and fields in a database system. Each attribute can have a descriptor associated with it to define the constraints on the values the attribute accepts. Each attribute can also have procedures or programs called daemons attached to it which are executed when the value of the attribute is modified. In such a system, a frame can inherit the attributes or properties of higher level frames.

A preferred embodiment of the present invention uses a frame-type representation in an object-oriented organization in which the frames represent objects. More specifically, the frames representing general concepts are referred to as classes and those representing specific occurrences of a concept are referred to as instances. In this context, attributes are termed members, and member inheritance and procedural attachment take place as in a frame system.

In object-oriented systems, however, objects communicate with one another by sending and receiving messages. When an object receives a message, it consults its predefined answers for messages to decide on what action to take. These answers can be stored directly with the object or inherited from a higher level object somewhere in the network hierarchy. Usually, the action involves triggering some rules, executing procedural code, or sending new messages to other objects in the system.

Similarly to the display platform user interface representation structures **118**, the application user interface representation structures **116** store descriptive information representative of the different objects that compose a user interface. Each object is described by a structure comprising a plurality of fields containing information representing an attribute of that object or a relationship between the object and another object. The user interface engine **117** maps each of the different objects that compose the user interface of a given application into the corresponding representations **118** in the user interface display platform **115** of choice for that application.

On the one hand, the user interface engine **117** requests application user interface representation structures **116** from the business process layer **120**. Once the business process layer **120** satisfies the request, the user interface engine **117** converts the application user interface representation structures **116** just received into user interface representation structures **118** that are expected by the user interface display platform **115** for display to the end user on a display station **111**.

On the other hand, when the end user performs an action through the display station **111**, such as selecting an item or modifying information, the user interface engine **117** translates that user request from user interface display platform representation structures **118** into the corresponding application user interface representation structures **116**, which are then handed to the business process layer **120** for execution of the end user request.

FIG. **5** is a schematic diagram of a sample mapping between application user interface representation structures **116** and display platform user interface representation structures **118**. In the figure, the user interface display platform **115** is exemplified as Microsoft Windows 3.x and the display platform user interface representation structures **117** are thus the internal Windows 3.x management structures. However, other user interface display platforms **115** using similar internal structures to manage windows are supported by the exact same user interface engine **117**. Notably, the internet's world-wide web, based on the HTML or Java user interface languages, is another example of user interface display platform **115**. Indeed, in a preferred embodiment of the present invention, the user interface engine **117** is written using Microsoft Visual C++ and based on the industry-standard Microsoft Foundations Classes (MFC) class library, which allows cross-platform development for Windows 3.x, Windows 95, Windows NT, MacOS, and UNIX-based user interface display platforms **115**, including internet web servers.

FIG. **6** is a block diagram of the operational modules of the user interface engine **117** of FIG. **4**. The user interface engine **117** includes an initialization module **117-1**, a user input module **117-2**, and a state router communications module **117-3**.

During initialization, the user interface engine **117** first initializes its initial state, setting up any structures necessary for operation. Depending on the implementation, the user interface engine **117** can then initialize communications with the business process layer **120**, receiving a client identification number. Depending on the implementation, the user interface engine **117** can also display an initial application menu or screen, initial objects that are provided by the business process layer **120**.

After completing the initialization, the user interface engine **117** continues to the user input module **117-2**. The user interface engine **117** waits for user input and processes it accordingly. In particular, the user input module **117-2** handles interactions with GUI objects and performs application-dependent actions in response to user inputs.

If the user performs an action that depends on remote processing, however, processing continues to the state router communications module **117-3**. In the state router communications module **117-3**, the user interface engine **117** creates outgoing application user interface representation structures **116** from the screen data and packs these structures for delivery to the business process layer **120**. Typically, the outgoing application user interface representation structures **116** contain values of screen fields which have changed since the previous call to the business process layer **120**. The packed application user interface representation structures **116** are then sent to the business process layer **120**, which returns packed application user interface representation structure **116** describing the result of the transaction. The packed application user interface representation structures **116** returned from the business process layer **120** are then unpacked and processed. Error messages can then be displayed, the screen can be updated with the results of the transaction, or a new screen can be shown. As long as the business process layer **120** does not indicate a fatal error, the

user interface engine **117** processing continues (resumes the wait for user input) at the user input module **117-2** until the user exits the application.

Most of the user interface engine **117** processing occurs in the handling of screens: building a screen from a description, processing application-updated values from the business process layer **120**, and sending user-updated values to the business process layer **120**. If a new screen is sent from the business process layer **120**, the current screen is discarded and replaced by the new screen. Communication with the business process layer **120**, and more specifically its main state router component (described below), is always initiated by the user interface engine **117** because a remote procedure call (RPC) mechanism which interfaces the user interface engine **117** with the business process layer **120** is preferably unidirectional and synchronous.

To simulate asynchronous communication using a unidirectional synchronous RPC model, the user interface engine **117** includes an ability to periodically poll the state router for messages during the user interface engine's **117** idle time, namely when there is no user input to be processed. This functionality is known as idle message polling.

Essentially, during idle message polling the user interface engine **117** queries the state router for any initial messages. At the start of an interactive application, a first screen needs to be displayed to the user. This screen is usually a sign-on, or logon, screen which contains fields for the user identifier and user password, with possibly peripheral buttons to change the user password and access help screens. In addition, other graphics, such as an application logo or wallpaper, might be decorating the screen. After these initial messages have been processed, resulting in the display of the logon screen, application menu, and other object for the user to act upon, the user interface engine **117** waits to process user inputs. If the user takes no action and idle message polling is enabled, the user interface engine **117** will periodically query the state router for any messages. If message polling is disabled, the user input loop will continue indefinitely. Using a window mapping structure, which is preferably a two-way associative array, it is possible for the user interface engine **117** to allow window control handlers of the user interface display platform **111** to manage general window operation and make callbacks to the user interface engine handlers when an action is required, for example, when a button is pressed.

In a preferred embodiment of the present invention, the user interface engine **117** can process any type of action from any type of screen object, e.g. a button being pressed, a control gaining the input focus, or the Tab key being pressed. Typically, when an action is performed, one of two things may happen: the user interface engine **117** performs some internal function based on the action, or sends information to be processed back to the business process layer **120**. In a particular preferred embodiment of the invention, all actions are referred back to the business process layer **120** for processing, along with any updated field values.

FIG. **7** is a schematic diagram illustrating the business process layer **120**, functionality layer **130**, and data access layer **140** of FIG. **1**. In a preferred embodiment of the invention, these layers can be hosted on similar platforms. In a preferred embodiment of the present invention, these platforms include a host processor **132**, in which the various engines are resident, internal or external storage **134**, on which the logic or data access server runtime environment resides, and a terminal console **136** which serves as a human interface for host administration purposes. In addition, a communications controller **138** such as a LAN controller, modem or similar device serves as an interface to a communication link. The host computer system **132** can be considered conventional in design and may, for example, take the form of a E55 workstation, manufactured by Hewlett Packard Corporation. As shown, the business process layer **120**, functionality layer **130**, and data access layer **140** further include a printer **135** which can be used to provide a permanent record of application log files, reports, source code, or process objects and flows according to the present invention.

FIG. **8** is a block diagram of the business process layer **120** of FIG. **1**. The main component of the business process layer **120** is a state router **122**. Conceptually, the state router **122** receives requests from the user interface engine **117** (FIG. **4**) and, based on the request, determines which actions to take. The state router **122** then calls upon the functionality layer **130** to perform the selected action, passing any required information. Upon completion of the action by the functionality layer **130**, the state router **122** accepts any resulting return information and forwards it to the user interface engine **117**.

The requests received from the user interface engine **117** include application user interface representation structures **121**. The application user interface representation structures **121** include request identifiers, transaction codes, screen information, and input/output buffers. A request identifier is the name of a function that needs to be executed in response to the request. There is one request identifier for any user interface event caused by the user. In this regard, request functions are similar to the conventional callbacks found in GUI languages such as X-Windows developed at the Massachusetts Institute of Technology, in Cambridge, Mass. Transaction codes are used to determine where to redirect the request. In this view, the state router **122** is simply a switch that differentiates between request identifiers and takes appropriate action in the form of a call to a function of the functionality layer **130**. Screen information is used to keep track of the current state of the application. Input buffers are used to carry information from the presentation layer **110** to the business process layer **120** and output buffers are used to carry information from the business process layer **120** to the presentation layer **110**.

FIG. **9** is a flow diagram of the communication mechanism between the user interface engine **117** and the state router **122**. As depicted, the user interface engine **117** includes a user interface routine **117-6** and initiates the communication by calling a pass message function **117-8**. The pass message function **117-8** first compresses the application user interface representation structures **116** to be transmitted into a single request string using a packing procedure. The request string compression performed by the packing procedure is necessary because the outgoing application user interface representation structures **116** cannot be transferred efficiently as such across the communication link.

The pass message routine **117-8** then calls a remote procedure call (RPC) routine **117-9** for actual transmission of the request string over the network. The RPC routine **117-9** takes two parameters: the request string to be passed from the user interface engine **117** to the state router **122** and the return string to be returned to the user interface engine **117** from the state router **122**. From the point of view of the state router **122**, requests arrive in the form of strings of characters that need to be decomposed into the logical components of the request. Consequently, the first step taken by the state router **122** is to decompress the request string into its logical components using an unpacking procedure.

The unpacking procedure converts the request string into an array of request application user interface representation structures **121**. This array is then passed to a main state router **122-1** function, which accounts for the core processing of the state router **122**. Routing logic **122-2** then directs the objects to servers in the functionality layer **130** or the database layer **140**. Once the state router **122** completes its processing, the resulting array of return application user interface representation structures **121** is again packed into a return string, which is passed back to the user interface engine **117** using an RPC mechanism **122-9**.

Because new application user interface representation structures **121** can be added to facilitate the transport of new types of objects as required by a particular application, the packing and unpacking functions include a library having primitives which pack and unpack bytes (8-bit integers), words (16bit integers), double words (32-bit integers), and strings (both variable- and fixed-length). To create a new application user interface representation structure **121**, a developer need only create packing and unpacking routines for that structure, assembling these functions from the primitive routines.

In the preferred embodiment of the present invention, the packing and unpacking library is written in such a way that the same source code compiles using structures (under ANSI C) or using object classes (under ANSI C++). Although the ANSI C language interface is very usable, the ANSI C++ language interface makes use of object-oriented features such as virtual functions to make packing and unpacking as transparent as possible. High-level packing and unpacking routines take arrays (or, in ANSI C++, containers) of application user interface representation structures **121** and create a single character string containing the packed information suitable for RPC transmission. This string contains type information as well as member data, so that any sequence of application user interface representation structures **121** can be sent and properly reconstructed at the receiving end.

In a preferred embodiment of the present invention, the state router **122** can be used to access the functionality layer **130** having custom-developed functionality servers as well as functionality servers converted from the IMS/VS model. The IMS/VS model is centered around the message concept, where the term "message" is used to refer to the model's communication structures with the functionality layer **130**. In a basic IMS/VS model, the state router **122** performs four main conceptual functions: log-on processing, IMS communication modeling, message conversion, and log-off processing. Log-on processing consists of checking the user authorization and issuing a client identifier. IMS communication modeling is decomposed into transaction routing, conversation management, and message formatting service. Transaction routing uses Input-Output Program Control Block (IO-PCB) and Alternate Program Control Block (ALT-PCB) IMS structures to route calls and messages between programs. Conversation management uses an IMS Scratch Pad Area (SPA) to store the processing context. Message formatting services uses Message Input Descriptor (MID) and Message Output Descriptor (MOD) IMS control blocks to format messages and screens. Message conversion performs message packing and unpacking. Log-off processing performs clean-up functions with commit point processing.

FIG. **10** is a flow diagram of the operations of a particular preferred state router **122** of FIG. **8**. Initially, when a logon request is received from the user interface engine **117** through the request user interface structure **121***a* and then authorized, the state router's **122** internal state is initialized with the current transaction code and the identifier of the first message. This message identifier is used to retrieve the full message format from a database repository **152**. This process will be discussed in further detail below.

The full message is placed in a message structure **122***a*. Among other pieces of information, a message format **122***a*-1, **122***a*-2 provides the identifier of the related screen as well as the identifier of the next message. The screen information associated with this screen identifier is then loaded from the database repository **152**. This screen information is passed back to the user interface engine **117** through the return application user interface representation structures **121**. The user interface engine **117** then displays the screen and awaits user input. When a users enters or changes data on a screen and presses a function key, the user interface engine **117** translates this user input into a request to the state router **122**.

Based on the contents of the request, the state router **122** saves its internal state into an old-state structure, and the current state is then updated with any new field values from the user interface engine **117**. State information is represented by a field state structure **122***b*. Then, the state router **122** determines which functionality server to call.

As mentioned previously, the request identifier, which describes a function to be executed, is included with the request from the user interface engine **117**. The state router **122** verifies the user's authorization to perform this function, and if authentication is successful, proceeds with its processing. If unsuccessful, an error condition is set, and the user is informed of an illegal access attempt. Assuming that authentication is successful, the determination of which functionality server to call is followed by a preparation of the message to be used for communication between the state router **122** and the selected functionality server, using a prepare to send message function **122***c* to build one or more messages to be put on the top of the outgoing Message Format Service (MFS) message queue **122***d*.

For an MFS-aware functionality server, the message is built as follows. Each field in the message, assuming field length n, is allocated n bytes in the message at a predefined offset. Additionally, a field may have two additional bytes allocated to it for passing the field's attribute in the message. A field's value is placed in the message if either of the following conditions holds: if the field's value has changed (which is determined by comparing the value of the field in the current state to the value of that field in the previous state), or if the field's attributes specify that its value should always be placed in the message regardless of whether it has been modified (this attribute is known as "pre-modified"). If the space in the message for a field value placed in the message exceeds the length of the field value itself, the allocated space is padded with the pad character specified for that field. Finally, if a field's value has not been modified, and the pre-modified attribute is not set for that field, its space in the message is filled with '@' characters. When all fields in the message have been considered, the message is complete. The message is then sent to the functionality server designated to handle the current transaction code.

The SPA **122***e* is provided as a functionality server-independent area of memory used for inter-functionality server communication. The SPA **122***e* is either newly-initialized, upon the first communication with the functionality server, or returned from the previous call to the functionality server.

When the call completes, a pointer is returned to the incoming MFS message queue **122***f*, where the functionality server placed one or more messages for transmission to the

state router **122**. The returned information includes an auxiliary buffer, a MOD structure, a SPA, and field value information similar to that of the message passed from the state router **122** to the functionality server. The auxiliary buffer specifies the name of the MOD structure. The MOD structure contains either a message identifier to describe the next message for updates to the current screen or a transaction name to initiate a switch to a new screen.

Upon receipt of this information, the state router **122** first determines whether the MOD structure contains a transaction or a message. If a transaction is present, indicating a screen switch, the new screen information is loaded from the database repository **152**, and its information is included in the return application user interface representation structures **121b** destined for the user interface engine **117**. The state router **122** then recalls the functionality server that corresponds to the new transaction in what is called an "immediate switch". On the other hand, if the MOD structure contains a new message name, the state router **122** retrieves the new message format from the database repository **152** and passes it to a prepare to receive message function **122*g***, along with the incoming MFS message queue **122*f***. This function updates the structures and processes the return message in a manner similar to the processing of the request message. Notably, new values and attributes are entered into the state router's **122** internal state. The attributes returned with each field specify whether the field is to maintain its old value, revert to its original attributes, or clear its value. The new values and attributes are then included in the return application user interface representation structures **121** array passed back to the RPC mechanism for return to the user interface engine **117**.

The above description of the state router **122** operations focused on the case of communication with functionality servers converted from the IMS/VS environment. In the case of custom functionality servers, the state router **122** still processes requests received from the user interface engine **117** in response to user interface event caused by the user in manner similar to that described earlier. However, much of the ensuing IMS/VS message processing can be bypassed in favor of a more generic mechanism embodied in the usage of a business process engine **124**.

In this custom model, the state router **122** still redirects processing to appropriate functionality servers based on the transaction codes received from the user interface engine **117**. However, when the functionality server returns, it passes back an event to the business process engine **124** component of the state router **122**. The business process engine **124** is an event handler implemented as a conventional non-deterministic finite automaton (NFA).

By way of background, an NFA is a mathematical model that consists of a set of states, a set of input symbols, a transition function that maps state-symbol pairs to sets of states, an initial state, and a set of final states. A special case of NFA is the deterministic finite automaton (DFA), which can have no unlabelled edges and at most one edge with the same label leaving a given state. Where time-space tradeoffs are an issue, an NFA is slower than a DFA but consumes much less space. In any event, an NFA and a DFA are both appropriate representations for real-life business processes. Furthermore, an NFA can automatically be converted into a DFA using fundamental principles of state machines and finite automaton theory. Consequently, which representation is used is of little consequence to a preferred embodiment of the business process engine **124**. In addition, because business processes can be composed of a series of unrelated processes or can even be decomposed into sub-processes, more than one NFA, possibly organized in a hierarchical fashion, can be used to represent the various business processes modeled by an application.

In any case, the business process engine **124** preferably implements an NFA as follows. Upon receiving an event from the state router **122**, the business process engine **124** first checks the validity of the event. If the event is valid, the business process engine **124** then examines all transitions out of the current state, which could be the initial state if this is the first call to the business process engine **124**. If the business process engine **124** does not find a transition that corresponds to the event received from the state router **122**, it simply remains in its current state, releasing control back to the state router **122**. On the other hand, should the business process engine **124** find a transition that includes the event received, the current state is saved and the next state is derived by starting at the current state and following the transition corresponding to the event received. The next state thus reached now becomes the current state.

Because states include initialization routines, the business process engine **124** executes any initialization routine associated with the next state immediately upon arrival at this next state. This initialization function can require another transition followed by another change of state, and therefore the business process engine **124** ends its processing by calling itself recursively, based on the event returned by the initialization function.

A complication can occur when a transition leads to a state that is not part of the business process modeled by the current NFA. In this instance, the business process engine **124** needs to switch to the NFA containing the next state. For this purpose, the business process engine **124** also maintains a current business process set. This enables the business process engine **124** to keep track of its position in the business process or NFA hierarchy.

FIG. **11** is a block diagram of the functionality layer **130** of FIG. 1. Conceptually, the functionality layer **130** manages the business objects manipulated by the business process layer **120**. These business objects constitute the fundamental components of an application. In a traditional manual system, a business object is associated with one or more physical paper forms. These forms contain the fields that hold the information relevant to the business object. Forms differ not only in their physical appearance, but also in the rules that govern their use. For example, a highly confidential form is treated differently from a non-confidential form. Other business rules may also govern the handling of forms. For example, some invoices might require more than one signature if their amount is bigger than a certain value. It is the form and the rules that govern its handling that define a business object in a traditional manual system.

The business objects represent the physical forms, the information in those forms, and the rules that govern these forms. As discussed previously in the context of a re-engineering or custom-developed application, the business process engine **124** of the business process layer **120** manages the flow of business objects, interprets their rules, and acts on these rules.

In a preferred embodiment of the present invention, the functionality layer **130** must also be able to handle IMS/VS COBOL functionality code. In IMS, functionality codes are structured into transactions composed of a main program called a driver and transaction programs for the various function keys the driver handles. Accordingly, a functionality server **135** performs a number of functions.

These functions comprise include file processing, server initialization, transaction call resolution, transaction entry

point processing, and server wrap-up. Include file processing comprises initializing global variables, notably the Program Specification Block (PSB) structures for each transaction. By way of background, a PSB defines, for a given transaction, the database which may be accessed, the database segments that are available, and the type of access (read, update, etc.) which may occur. A PSB is also a collection of Program Communication Blocks (PCB). A PCB is an IMS structure to control the access to data as will be described in detail below.

Server initialization comprises unpacking the message received from the state router **122**, to obtain the SPA and its call parameters, and assigning local function pointers. An additional level of indirection for each transaction program main routine is necessary for ANSI C to mimic the COBOL "goto" capability to span across routines and exit at any point in the program.

Transaction call resolution comprises determining the driver to call based on the transaction identifier specified in the RPC received from state router **122**, associating the appropriate PSB structure obtained from the include file with the selected driver, and calling the driver with its arguments. Once in the driver code, control flows according to defined COBOL language principles. Two COBOL constructs merit further description in the context of ANSI C functionality code converted from COBOL, namely special routines called entry points and COBOL variables.

Transaction entry point processing comprises performing calls to transaction entry points in the converted COBOL functionality code. Transaction entry points include routines to perform calls to various local (sub) routines or external COBOL library routines, as well as routines to establish communication with the data access layer **140** (ENTRY statement) and perform calls to the database server/IO devices (CBLTDLI). A typical database access consists of an initial GET call to populate the I/O work area, followed by modifications to the I/O work area for subsequent insert (ISRT) or replace (REPL) calls. After each database access, the return status of the call is checked to take action based on the result of the database operation.

In a preferred embodiment of the invention, COBOL variables are implemented as COBOL structures. A COBOL structure contains a data field, which consists of a pointer to an area in memory used to store the value of the variable. Another field stores the length of this data memory area. The COBOL structures also contain a mask, which is a string describing the COBOL format specification for the variable. For example, a mask of "PIC X(3)" refers to a string of 3 characters. Sometimes COBOL variables exist in a hierarchy of other COBOL variables. Consequently, the COBOL structures also contain the number of sub-variables or sub COBOL structures in the current variable or COBOL structure hierarchy. A COBOL structure variable can thus be viewed as pointing to a buffer containing its data, all sub COBOL structure variables pointing to the proper offsets in that buffer.

The last function performed by the functionality layer **130** is server wrap-up. Server wrap-up includes preparing return data, notably packing the SPAM for return to the state router **122**.

FIG. **12** is a block diagram of the data access layer **140** of FIG. 1. The data access layer **140** comprises a number of data servers. As mentioned previously, the data servers are used to access and retrieve data from the data storage layer **150**. Preferably, one data server exists for each of the four application object repositories. Consequently, there is user interface data server **141** to manipulate user interface objects **142**, a business process data server **143** for business processes **144**, a business object data server **145** for business objects **146**, and a database server **147** for application data records **148**. The data servers constitute the sole interface between the data storage layer **140** and the functionality layer **130**, and each data server is only in charge of exchanging with the functionality layer **130** information about the type of application object it services.

According to client/server technology, a server is any program that runs a function invoked by a different program. A server is thus a software concept that provides a way to package together a set of related functions. Consequently, it could be implemented as a conventional third generation language sub-routine or library.

In the present invention, a number of the components referred to as libraries can be implemented as servers and vice versa. The server implementation, however, is better suited to inter-platform communication, and is a preferred embodiment for the communication links of the present invention.

By way of background, servers can be left to run continuously in stand-alone mode and accept requests from multiple clients. The set of functions, or services, provided by a server constitutes the server interface. This interface is specified in an Interface Definition Language (IDL) file. The concept of servers is well known, and details of server development and operations, including stub generation and IDL file syntax and specification are commercially available.

The data access layer **140** can now be viewed as a set of database access and retrieval functions. There is one function for each one of the data access construct of the source Data Base Management System (DBMS). Each such function must emulate the behavior of the corresponding source DBMS data accessor construct. In the preferred embodiment of the present invention, the target DBMS is typically based on a conventional relational model, and is called a Relational DBMS (RDBMS). The source DBMS can be built around a number of conventional data models. The most common such models are the flat file, hierarchical, network, CODASYL (Conference on Data Systems Languages), relational, and object-oriented data models.

By way of background, the flat file model is a generalized file management system that adds report generation and file management additions to third-generation programming languages such as COBOL. An example of such as model is the Report Program Generator (RPG) from International Business Machines, Incorporated. The hierarchical model is a hierarchical tree of nodes made of records and fields, with tree search capabilities. An example of such a model is IMS Data Language 1 (DL/1). The network model is an extension of the hierarchical model, where nodes may have more than one parent. The network model is also referred to as CODASYL, which is the initial embodiment of this model using owner and member records linked by pointers, as originated by Honeywell Information Systems, Incorporated. Examples of the network model are Integrated Data Store (IDS) from IBM, and Integrated Database Management System (IDMS) from Cullinet, Incorporated. Given the fundamental differences between these models, the data access library must be careful to map the semantics of the source database to the appropriate constructs in the target database.

In its simplest form, this mapping results in a simulation of the source DBMS in a relational model, which is not always ideal for readability and maintainability. A preferred

embodiment of this invention extends this mapping to a true conversion of the source data model to the relational data model. The initial relational model thus obtained is further normalized to a specified degree controlled by parameter using a bacchus-normal form utility, leading to a true relational model. In the context of the present invention, such libraries exist for all the common source data models mentioned. Because an overwhelming majority of existing database systems fall into one of the source data models above, it is likely that most existing applications can be handled by one of these libraries with minor or no changes.

In a preferred embodiment of the present invention, the source DBMS is IMS DL/1. As discussed previously, IMS functionality code is structured into transactions composed of a main program called a driver and subprograms for the various function keys the driver handles. For every such IMS transaction in the functionality server, an entry function is called once to initialize the database server. This entry function calls a database server function through an intermediary to initialize the PCB structure in the database server. Then, every time the functionality server needs to access the database server, it calls the function CBLTDLI( ), which in turn calls a database server function to call the appropriate database function (e.g., GET, INSERT, DELETE, and REPLACE primarily). Communication between the functionality server and the database server is thus reduced to two functions: init and CBLTDLI (which, in IMS, stands for CoBoL To Data Language 1).

The main function of a database server is to fulfill requests for data from the functionality server **135**. In the preferred embodiment of the present invention, this includes implementing the IMS DL/1 CBLTDLI call. This can be decomposed into three tasks: server initialization, CBLTDLI call resolution, and database access function execution. As discussed earlier, server initialization is performed on the PCBs for the current transaction. Then, the database server resolves CBLTDLI database calls.

The database server communicates with the database through ANSI SQL queries. The database server implements each type of IMS DL/1 function as follows. The database server first validates the arguments to the IMS DL/1 function. It then dynamically creates an ANSI SQL query string. In the preferred embodiment of the present invention, this query string is forwarded to the database using the Oracle Call Interface (OCI) from Oracle Corporation. At a high-level, the process consists in initializing bind and define variables, setting currency information, executing the SQL query, and returning query status.

FIG. **13** is a block diagram of the data storage layer **150** of FIG. **1**. The data storage layer **150** is a repository for data accessed by the data access layer **140**. The user interface data repository **152** provides user interface objects **153** to the data access layer **140**. The business process data repository **154** provides business processes **155** to the data access layer **140**. The business object data repository **156** provides business objects **157** to the data access layer **140**. The application data records repository **158** provides data records **159** to the data access layer **140**.

FIG. **14** is a schematic diagram of a hardware platform for the data storage layer **150** of FIG. **1**. As shown, the data storage layer **150** is hosted on a platform that includes a host processor **151**, with one or more Central Processing Units (CPU), in which the Data Base Management System (DBMS) is resident, internal or external storage **153**, usually an array of mirrored disks, on which all database data resides, and a terminal console **155** which serves as a human interface for host administration purposes. DBMS log files are stored in storage unit. A printer **157** is usually present for database diagnostics or large data outputs. In addition, a LAN controller, modem or similar device serves as a communication link **159**. The DBMS and the host computer system **151** can be considered conventional in design and may, for example, take the form of an Oracle relational DBMS, manufactured by Oracle Corporation, and a T500 mini-computer, manufactured by Hewlett Packard Corporation respectively.

A preferred embodiment of the present invention uses the relational data model for the data storage layer. The relational data model can be viewed at three different levels: conceptual, logical, and physical. The conceptual level consists of entities, attributes, and relations. Entities are things that exist on their own, distinguishable from other objects. Entities (records, rows) are described in terms of their attributes (fields, columns). Relations are common fields between entities used to connect entities together. The Entity-Relationship data model (ER) is the predominant conceptual level description tool. It is used as a diagramming technique where rectangles represent entities, circles represent attributes, diamonds represent relationships. The logical level consists of records, fields, and relations.

The schema is the logical level description tool. In the relational model, a database schema consists of the description of the tables, their fields, and the fields formats and domains. The relational algebra provides the theoretical basis for the model, with five operators: selection, projection (deleting columns from table), product, union (adding the rows of two tables), difference and a composite: join.

Query languages based on relational algebra are Structured Query Language (SQL) and Query By Example (QBE), both originated by IBM. SQL is usually embedded within a third generation language such as C, called the host language. Because host languages do not typically have multi-record operations, special SQL commands such as the cursor concept are provided to process multi-row query results in a record-at-a-time fashion. A cursor is a name given to a query. When a cursor is opened, the corresponding query is executed. Any subsequent fetch command on the cursor returns a new row to the host language. When the cursor is closed, query results are no longer available to the host language. Other special commands in SQL embedded mode include transaction processing features, dynamic SQL generation, and authorization control. The physical level deals with data dictionaries, data definitions (physical file structures, file space allocation), storage devices (data compression), access methods (sequential, index-sequential, direct).

Sequential files use a sorted column to perform sequential search. Index-sequential adds an index to a given column to provide random access. Large indices may themselves be indexed. Sequential files are difficult to maintain because adding or deleting a record requires reorganization of the whole file. Indexed (inverted) files remove the sequential part. Fully inverted refers to indices associated with each column. Indices carry update overhead but enable fast access. Direct access uses a single key and a calculation from that key to locate the physical address of the data in memory.

The distribution discussed in the context of the present invention is reduced to process distribution. However, the data storage layer can also be distributed among different platforms. A decision on how the data can be distributed depends on the following four criteria: connectivity, volume,

volatility, and usage. The inter-connectivity of the data is defined by the relations between entities. The volume or size of each entity is evaluated in terms of memory usage as well as number of records. The volatility is the rate of change in the volume of each entity. Finally, the usage is determined by the frequency of use of the various screens and the transaction rate.

FIG. **15** is a block diagram of the control layer **160** of FIG. **1**. The control layer **160** includes utilities for transaction management **161**, security **162**, system administration **163**, server management **164**, accounting **165**, network management **166**, and configuration management **167**. These utilities can take the form of libraries or can consist of graphical user interfaces.

Transaction management **161** provides library utilities to manipulate transactions. Transactions are a way to package related application elements together. Transaction processing refers to the manipulation of groups of elements as opposed to the manipulation of the individual elements. A transaction that successfully completes the processing of all the elements that compose the transaction is finalized by a database commit, which saves the results of the processing of all its elements in a permanent form, usually in the database. Should the processing of one of the transaction elements fail, the transaction elements that have already been committed to the database need to be un-committed, and the whole transaction is rolled back, with appropriate error messages posted. Transaction management **161** is useful in distributed systems to insure data consistency in the absence of user-defined integrity constraints.

Security **162** is provided at all layers of an application through a library of functions to manage system as well as data security. At the presentation layer level **110**, security functions are available at logon time and at the menu, screen, field, and button levels. At the functionality layer **130** level, access to entire servers or to a subset of the services provided by a server can be restricted. At the data layer **140** level, security functions manage access control lists (ACL), which enable application administrators to set up a hierarchy of user types for controlling access to application resources. In addition, the underlying database management system usually provides a wide range of security features which are implicitly available to the application.

System administration **163** provides graphic facilities for administrators to perform basic application administration tasks, such as lookup table maintenance, user access maintenance, and application backup and recovery.

Server management **164** provides mechanisms to organize servers in a hierarchical model in which entities called brokers keep track of the servers available to a given client and of their location. This can increase the robustness of applications through server redundancy.

Accounting libraries **165** are available to perform logging of application operations at all application tiers for performance monitoring, auditing, or error tracing and recovery. Logging is also available on a per client, server, or broker basis. Logging can come in various flavors, with control over the level of detail provided or the application resource or component being monitored.

Network management **166** provides a graphical interface to monitor clients, servers, and brokers. Network traffic and performance can thus be monitored, and network components restarted automatically or manually upon failure.

Configuration management **167** provides libraries for a number of diverse purposes. For instance, application version management functions are provided. In addition, currency is handled through locking functions to insure data consistency. Data integrity is controllable at the functionality layer by the business objects rules or constraints. The underlying database management system usually provides data integrity controls implicitly available to applications, but the usage of such controls are not recommended because it would mean encoding business logic in the data layer **150** instead of confining it to the functionality layer **130**, and would therefore be contrary to the fundamental principle of the preferred multi-tiered architecture.

Communication links **170** are used by the other components to exchange information by the way of computer networks. By the way of background, computer networks consist of interconnected collections of autonomous computers. Networks are usually described in terms of the well known International Standards Organization (ISO) Open Systems Interconnection (OSI) reference model. ISO OSI describes networking in terms of seven layers, from lowest to highest: physical, data link, network, transport, session, presentation, and application.

Because distributed systems are a special case of a network that is transparent to the application, the present invention can rely on any one of the prevalent networking standards. For example, using ISO OSI terminology, one embodiment of the present invention can be based on the internet de facto standard. At the lowest physical and data link levels, a preferred embodiment of the present invention can be a standard communication media, such as a telephone network, local area network (LAN), wide area network (WAN) or direct line. The Internet Protocol (IP) can be used as the network protocol and the Transmission Control Protocol (TCP) as the transport protocol. Session and presentation layers do not exist in the internet model. Application protocols include FTP for file transfer, SMTP for electronic mail, and TELNET for remote login.

Because distributed applications are network transparent, the distinction lies in the software. Therefore, the preferred embodiment of the present invention uses commercial tools that hide all of the underlying network complexities and isolate applications from network implementations. These tools are based on the Remote Procedure Call (RPC) operating over TCP/IP or over the Distributed Computing Architecture (DCE) mechanism. Using an RPC, a client program performs what looks like a conventional function call. A piece of RPC generated code called a client stub handles all the aspects of handling the call, including packaging the function arguments for transport, called marshaling, and carrying out the transport.

On the server side, a similar RPC server stub unpacks the function arguments, called unmarshaling, and passes them to the server code that implements in a conventional way the function requested by the client. Upon completing the execution of the requested function, the server returns the results of the call to the client in a similar way. Because all transport complexities are addressed by the RPC generated stubs, RPCs acts just like conventional local calls.

For more complex distributed applications, a preferred embodiment of the present invention can use tools based on DCE, which is a more comprehensive distributed system infrastructure that includes directory services, distributed security, multi-threading, distributed file system, and central time management, in addition to RPCs. As sample implementation of RPCs effective over both TCP/IP and DCE networks, is the Entera toolkit from Open Environment Corporation.

Returning to FIG. **1**, the re-architecting system **20** includes a user interface conversion utility **210**, a procedural

language conversion utility **220**, and a data definition language conversion utility **230**.

FIG. **16** is a block diagram of the user interface conversion utility **210** of FIG. **1**. The user interface conversion utility **210** converts the user interface of an existing application represented by the source user interface definitions **211** into target user interface definitions **213** using the user interface converter **212**. In a preferred embodiment of the present invention, the source user interface definitions **211** can be viewed as IMS/VS Message Format Service (MFS) files.

Screen definitions provide structural information about the fields that compose the screen layout. Message definitions provide content information about the fields, such as the data they contain and their attributes (protected, highlighted, etc.).

Target user interface definitions **213** can take one of three forms: database files **246**, a header file **247**, and a GUI file **248**. Database files **246** contain the set of statements necessary to populate user interface repository **152** with screen and message information for MFS file **211**. In a preferred embodiment of the present invention, the database files **246** can be viewed as ANSI SQL scripts.

A deletion script removes from the user interface repository **152** any definitions for the MFS file **211**. Once this repository cleanup is accomplished, an insertion script adds to the user interface repository **152** any new definitions for the MFS file **211**. Consequently, the user interface conversion utility **210** can be run multiple times for the same MFS file without negative effects. In a preferred embodiment of the present invention, the user interface repository **152** is a standard RDBMS such as Oracle Server **7** from Oracle Corporation.

Information stored in the user interface repository **152** is converted at application runtime into the user interface representation structures of the presentation layer **116**. The user interface engine **117** of the presentation layer **110** then maps application user interface representation structures **116** into display platform user interface representation structures **118**, used by the user interface display platform **115** for display to the user. Accordingly, target user interface definitions **213** effectively constitute an intermediary user interface definition language for storage of user interface information in the user interface repository **152** and eventual user interface representation structures **118**.

Header files **247** are an alternative to database files **246**. In a preferred embodiment of the present invention, user interface representations are stored in the user interface repository **152** and retrieved as needed from this repository by the business process layer **120**. This is an appropriate mode of storing a large amount of user interface representations on a back-end database host, thus alleviating performance and space constraint problems on the client or business process hosts. However, for smaller applications, user interface representations may not need to be stored on a separate user interface repository **152**. Accordingly, a user interface converter **212** can generate a header file **247** instead of database files **246**.

Such a header file **247** is then passed to the business process layer **120** to provide information necessary during application runtime operations. In this scenario, the header file **247** can be viewed as declarative statements in ANSI C that are compiled as part of the source code for the business process layer **120**.

GUI files **248** are used by application developers and maintenance personnel to modify application screens and messages as part of the re-engineering system **30**. In a preferred embodiment of the present invention, the GUI file **248** are written in Microsoft Visual Basic. The application re-engineering process **30** then uses the GUI file to load screen information in Visual Basic, which can be viewed as the graphical user interface editor **310**, make any modification in Visual Basic, resulting in a modified GUI file, and then run a Visual Basic to Oracle conversion process as described regarding the graphical user interface editor **310** to load the modified GUI file into the user interface repository database **152**, ready for usage by the application.

The user interface conversion utility **210** calls the user interface converter **212** to generate the "target" representation just described. In a preferred embodiment of the present invention, the user interface converter **212** is an ANSI C program, which takes a MFS file as an input and generates output files. To perform this function, the user interface converter **212** can be structured using conventional compiler technology, including a scanner **241**, a parser **243**, and a code generator **245**.

Specifically, the scanner **241** reads characters from the MFS file until a token has been read. The scanner **241** adds the token just obtained to a symbol table in which all the identifiers of the source language are stored, along with their characteristics. The tokens are then passed to the parser **243**.

The parser **243** can be viewed as a function that parses the statements of the source language. In this context, the delimiter that enables the user interface converter **212** to determine when the end of a statement has been reached is defined by the particular syntax of the source. Once a statement has been parsed, the parser **243** calls another function, which returns the type of the statement that has just been parsed. This type is then passed to the code generator **245**.

The code generator **245** can be viewed as a large switch that, given a statement type, calls the appropriate function to generate "target" code for this statement. The code generator **245** relies on a library of functions that handle the actual code generation for the entire set of statements of the source language. One such library exists for each source language, with ANSI SQL, ANSI C, and Microsoft Visual Basic as the target languages.

FIG. **17** is a flow diagram of the procedural language conversion utility **220** of FIG. **1**. The procedural language conversion utility **220** converts the functionality and data access programs of an existing application into the programming language targeted for the implementation of the functionality layer **130** (FIG. **1**). This conversion process consists of two main phases. A first phase (Phase A) converts the source language **221** into an intermediary language **225**. A second phase (Phase B) then transforms the intermediary language **225** into the final target language **227**.

The first transformation is executed by a first phase (Phase A) transformer **224**, customized for each source language environment **221**. The intermediary language **225** is a meta language designed to facilitate application maintenance.

The meta language is independent of source and target languages and supports the use of an independent data dictionary **228**. The data dictionary **228** generation is preferably achieved through the use of data population tools **223**, which use constructs in the source code and related data files.

The meta language is used as the target for the conversion of numerous source languages, and in turn can be transformed into multiple different target languages. The meta language is converted to a target language using a second

phase (Phase B) transformer **226**. There is a specially customized second phase transformer program **226** for each target language environment **227**. The procedural language conversion utility **220** as a whole is applicable to batch as well as on-line applications.

FIG. **18** is an exemplary source code fragment **221'**. As illustrated, the source code fragment includes assignment statements **221-A, 221-C, 221-E, 221-F, 221-H**, conditional-branch statements **221'-B, 221'-G, 221'-J, 221-K, 221-M**, jumps, **221-D, 221-L** function calls **221-I, 221-N**, and labels **221-0**. The assignment and conditional branch statements can use variable values or literals.

FIG. **19** is a block diagram of the first phase transformer program **224** of FIG. **17**. As shown, the main elements of this transformer are a grammar definition for the source language **251**, a dynamic symbol table generator for the source language **252**, and a number of rules **253** for transforming a source language **221** to the meta language **225**. In addition, an external data dictionary **228** contains the data structures, definitions, and common logic constructs pertaining to the source application.

In terms of control flow, the first phase transformer **224** receives a file of source language code **221** as input. This file is then semantically parsed using the source language grammar definition **251**. A full grammar for the source language is specified using a programmatic grammar encoding scheme.

The symbol table **252** is dynamically generated during parsing based on data definitions found in the source code **221** as well as in the data dictionary **228**. All relevant data (i.e. non-procedural) elements found in the source code **221** are incorporated into the symbol table **252**, while irrelevant elements are discarded based on source language grammar **251**. The symbol table **252** is thus dynamically built to contain symbols for all data elements, data structures, data definitions and variables relevant to the source code file **221** being transformed.

FIG. **20** illustrates in FIGS. **20A–20B** a parse tree for the source code fragment of FIG. **18**. As illustrated, the source code **221** is parsed into a series of statement lists **225**. Each statement list includes at least one statement. A statement can include an additional statement list. For example, the first instruction **221-A** (FIG. **18**) is parsed into an assignment (ASSIGN) which assigns the variable (VAR) LOW-VALUE to the variable list (VARLIST) of variables (VAR) YMS-CALL-RESULT and FLAG-AREA. Each source language instruction **221-A, . . . , 221-0** of FIG. **18** is represented as a respective branch cluster **225-A, . . . , 225-0** on the parse tree **225** of FIG. **20**. The branch clusters are on branches **225-1, . . . , 225-8** of the parse tree **225**. From this representation, the source language is converted to the intermediate language.

Once parsing is complete, a complete set of rules **253** for transforming source language code to meta language code is programmed as declarative rules in a separate conversion routine **253**. The conversion rules **253** operate as follows. A rule is applied to each source language construct parsed using the grammar **251**. When a rule is executed, a source language construct is transformed to its equivalent construct in the meta language. Procedural constructs and primitives are thus regenerated in the meta language. Operations on data elements and data structures are first validated using the symbol table **252**. Based on this validation, an equivalent operation is custom-generated in the meta language. The rule being executed for this validation operation generates a specific construct to reproduce the semantics of the transformed operation, such as a conversion from integer to floating point, if required.

In addition to procedural constructs and data elements and structures, a subset of the rules specializes in the extraction and transformation of external data access commands into application-independent external data access commands. In a preferred embodiment of the present invention, application-independent external data access commands could be encoded using the ANSI-SQL language. External data access operations are thus parsed and transformed into separate, application-independent data access commands. These generated data access commands are stored in at least one separate output file.

FIG. **21** is an intermediate language file **225'** for the source code fragment of FIG. **18**. The file **225'** is created by traversing the parse tree **225** depth-first, left-to-right. A node is not output to the file **225'** until all children are output. This technique results in a reverse polish or postfix notation. As illustrated, each branch **225-1, . . . , 225-8** of the parse tree **225** is represented as a statement **225-1', . . . , 225-8'** in the file **225'**. The trunk nodes of the parse tree **225** are represented as a terminal statement **225-9'**.

In summary, executing the transformation rules on the source code produces two separate outputs: the transformed source code in a meta language, which is constituted of procedural source code and data definition structures, and a set of data access command.

FIG. **22** is a block diagram of the second phase transformer program **226** of FIG. **17**. Similar to the first phase transformation, the main elements of the second phase transformer **226** are a grammar definition **261** for the meta language **225**, a dynamic symbol table generator **262** for the meta language **225**, and a number of rules **263** for transforming the meta language **225** to a target language **227**. In addition, the second phase transformer program **226** uses the same external data dictionary **228** as does the first phase transformer program **224**.

In terms of control flow, the second phase transformer **226** receives as input meta language program files **225**, usually constituted by meta language procedural source code and data definition structures files, as well as data access command files, both generated by the first phase transformer **224**. The meta language program files **225** are then semantically parsed using the meta language grammar definition **261**. A full grammar for the meta language is specified using a programmatic grammar encoding scheme.

The symbol table **262** is dynamically generated during parsing based on data definitions found in the meta code **225** as well as in the data dictionary **228**.

The data access command files portions of the meta language program files **225** provided as input are parsed using a different mechanism. Because this mechanism is a simplified version of the one used for procedural source code and because it is based on the exact same principles, it will not be detailed any further here.

Upon completion of parsing, a complete set of rules **263** for transforming the meta language code **225** to the target language code **227** is programmed as declarative rules in a separate conversion module **263**. The conversion rules **263** includes rules for transforming meta language data definitions into target language data structures, rules for transforming meta language variable definitions into target language variables, rules for generating target language initialization routines for the data structures and variables mentioned above, rules for transforming procedural meta language source code into target language source code, and

optionally rules for customizing the application-independent data access commands for a particular application such as Oracle. A rule is applied to each language construct parsed using the meta language grammar **261**.

Language constructs include: data definitions, variable definitions, and procedural constructs. Data definitions in meta language code are transformed into target language data structures by executing a set of rules written for this purpose. Variable definitions in meta language are also transformed into target language variables by executing a set of rules written for that purpose. For each variable definition created, an initialization routine is created. Procedural constructs are transformed into their equivalent in the target language. Data access commands can be tailored for a particular application as required by the target environment. Executing the transformation rules on the meta language source code produces output files containing the transformed code in the target language: procedural source code files, data definition structure files, and final data access command files.

The target source code **227** includes calls to specially designed and implemented libraries to support functionality that is not provided natively in the new target environment. These functions include variable handling, value assignment, data access services, transaction processing, and other. Depending on the target environment, different runtime libraries are required in order to guarantee correct execution in the new environment. The scope of these libraries is determined not only by the target environment, but also by the source environment. All source constructs must be mapped to equivalent constructs in the target environment.

FIGS. **23A–23B** are C and C++ target code fragements **227'**, **227"**, respectively, for the source code fragment of FIG. **18**. As illustrated, the intermediate language illustrated in the output file **225'** of FIG. **21** is transformed into a select target language source code **227**. The target language can be any procedural programming language (such as C) or any object-oriented programming language (such as C++). As described, a different second phase transformer program **226** is used to generate source code in each target language.

Together with the transformation of the code to the target environment, the transformation process permits new data organization methods. If new methods are used to organize data, additional libraries might be required to achieve complete compliance with the original application behavior. An example of specialized library for IMS/VS data organization method was outlined above in the context of the data access layer **140**.

In addition to the transformed code and required libraries, a runtime infrastructure must be provided for application execution. The infrastructure in question corresponds to the multi-tiered architecture **10** detailed above. Because the description of the architecture **10** focused on on-line programs, a number of key differences should be noted in its application to the batch components of applications. Batch-related code modules only interact with their calling process, eliminating the need to maintain a two-way communication structure with the user interface module and the accompanying state information in the business layer. Instead, re-architected batch processes are stand-alone programs constituted by a wrapper that provides means to parse the input arguments and call the top-level batch job. Typically, this top-level batch job requires some form of job scheduling infrastructure. As an example, legacy Job Control Language (JCL) can be converted to a scripting language-equivalent such as UNIX shell, Perl, or REXX. The resulting script then calls the various batch programs, interleaved with scripting commands or library calls that provide functionality that is similar to that of the source legacy system. In spite of these differences, batch conversion and processing follows the same fundamental principles as on-line, in a simplified manner. Consequently, a full discussion of the specifics of batch conversion and runtime execution will not be detailed any further.

FIG. **24** is a block diagram of the data definition language conversion utility **230** of FIG. **1**. The database conversion utility **230** is used to convert a source database language **231** into a target database language **237** using a database converter **234**. As illustrated, the conversion must address the database structure, encoded using a Data Definition Language (DDL), and referred to as a schema, as well as the database data.

In a preferred embodiment of the present invention, the target DDL **238** is a relational database schema specified using conventional ANSI SQL language. Such a schema defines the tables that compose an application, along with their key fields, and other descriptive fields. Initial values and other constraints such as referential integrity clauses may also be included in this schema. Because relational schemas are well understood, and ANSI SQL syntax is well-documented, the primary task of the DDL converter **235** is to map the syntax of source DDL **232** to the corresponding ANSI SQL syntax. In a preferred embodiment of the present invention, this "target" DDL **238** can be viewed as an intermediary language that can then be converted to the final target DDL language for increased maintainability and flexibility, as was the case with the user interface and procedural language conversion utility. For illustrative purposes, IMS DL/1 can be considered as the source DLL **232**.

FIG. **25** is a block diagram illustrating a schema conversion. As shown, the DDL converter **235** is sub-divided into a first converter **235***a* and a second converter **232***b*. The first converter **235***a* takes DBDxx **232***a*, DBDxx **232***b*, and COPYLIB **235***c* as inputs and generates a table creation SQL statement file **238***a*, an index creation SQL statement file **238***b*, a primary key creation SQL statement file **238***c*, and a database schema ANSI C header file **238***d*.

By way of background, an IMS database schema depends on Data Base Descriptions (DBD) and COBOL copy libraries (COPYLIB). All IMS data bases must be defined through DBD generation prior to use by application programs. A DBD is the DL/1 control block that contains all the information necessary to describe a data base, namely segment types, physical and logical relationships between segment types, database organization and access method, and physical characteristics of the database. COPYLIB contains COBOL data structures definition and is used to create corresponding C structures and IMS segment definitions.

The first converter **235***a* generates a table file **238***a*, which is used to create tables in the target RDBMS. The table file **238***a* includes simple relational table creation statements, without indices, keys, or reference integrity. Consequently, it an be generated directly from COPYLIB **232***c* information, without any DBD input. For example, the COPYLIB entry for a segment is used to generate a corresponding relational table.

A relational table is build from an IMS segment as follows. First, all the key fields of all the ancestors of the segment in question in the IMS hierarchy are included in the relational table into which the segment in question is being

converted. Then, all the local key fields of the segment being converted are included in the target relational table. Finally, all the local non-key fields of the segment to convert are added to the target relational table.

The first converter **235***a* also generates the index file **238***b* and the primary key file **238***c*, which are derived from the table file **238***a*. The first converter **235***a* creates one index for each parent of the converted segment by concatenating the keys for that parent. One index is also created for each local key field.

The first converter **235***a* also creates the schema header file **238***d* having information for each segment using the DBDxx **232***a* and corresponding DBDxxL **232***b* to provide schema information to application programs. Preferably, a segment header file includes two ANSI C data structures: a segment and a segment array. The segment structure includes the following information: a segment name, the names of the segment columns, the segment child index in the form of another header file, the length of each segment column, the expanded column length, the PIC mask for each segment column indicating column type and size, the column usage, a logical key and the corresponding local key index, the number of columns in the segment, the number of parent keys in the segment, and the number of local keys for this segment. The segment array structure includes the following information: a segment name, physical child names, the number of children, a logical flag, and a pointer to the corresponding segment data structure.

In addition to the first converter **235***a*, the second converter **235***b* is used to convert PSB definitions **232***d* into PSB header files **238***e*. As mentioned previously, a PSB defines the database which can be accessed, the segments within the database which are available, and the type of access (read, update, etc.) which can occur. The PSB header file **238***e* provides such database access information to application programs.

FIG. **26** is a block diagram of a converter **235***a*, **235***b* of FIG. **25**. Both converters **235***a*, **235***b* are built using the conventional principles of compiler design. This includes a scanner **271** to decompose source data definition language **232** into tokens **272**, a parser **273** to assemble tokens **272** into a syntactic parse tree **274**, and a code generator **275** to generate target data definition language constructs **238** from the parse tree **274**. This technology is well-understood and documented in existing literature and the details of a particular compiler are highly dependent on the source and target languages. Consequently, the converters **235***a*, **235***b* will not be detailed any further.

Once the database structure is converted, the next task is to convert and load database data. In the IMS example, source data **233** is stored in a hierarchical fashion. The first converter **235***a* generates a data loading pattern file DBDxx.dlp **238***f* for a given physical DBD using DBDxx **232***a* and COPYLIB **232***c* information.

FIG. **27** is a block diagram of the data converter **236** of FIG. **24**. A DBDxx.dlp file **231***a* is provided as input to the data converter **236**, along with a flat file **231***b* that contains the data to be converted. The data converter **236** uses the data loading pattern specified in the DBDXX.dlp file **231***a* to determine the desired target format for each data record in the data flat file **231***b* and generate an ANSI SQL file **239***a* containing the INSERT statements necessary to populate the target database with the data provided.

The data converter **236** needs to take into account a number of technical issues in performing its task on IMS data, including packed data handling and field redefinition. In IMS, some numerical field are compressed into a binary representation for storage efficiency. Such packed data still present in binary format in data flat files **231***b*, is first unpacked and then moved to the end of the record, before using a filler, namely clearing the packed data original positions with blanks. When a data loading pattern file **231***a* specifies packed data fields, the data converter **236** searches the end of the record instead of the original field position to locate the proper data in unpacked format.

Another peculiar situation arises with redefined fields. COPYLIB **232***c* sometimes specifies a redefinition for one or more fields. Such re-definitions are ignored and left to the functionality code to handle. When redefined fields include packed data however, the data converter **236** creates one filler for each redefine after moving all unpacked data for the redefine in question to the end of record. Combination between re-definitions and packed data fields is thus treated as a special case of the latter.

Once all conversion is complete and all output files are available, the order of creation for a given target database table is first to create the table using the table creation file **238***a*, next load the data from the target data **239**, then create the primary key using the primary key creation file **238***c*, and finally to create the indices from the index creation file **238***b*. Once the target database structure is established and all database data is loaded, the ANSI C structures in the schema header file **238***d* and the PSB header file **238***e* are used at runtime by application programs to access the target database structures.

As mentioned previously with regard to FIG. **1**, the custom and re-engineering system **30** focuses on providing an enterprise a facility for maintaining and enhancing distributed infrastructure. Even though this facility is an integral part of the overall system of the present invention, it is really an add-on facility that becomes paramount once the transition is complete. Consequently, only an overview of the custom and re-engineering system **30** will be provided here. At a high-level therefore, the custom and re-engineering system **30** includes a graphical user interface editor **310**, a graphical business process editor **320**, a graphical business object editor **330**, a graphical data record editor **340**, a logic development environment **350**, and facilitation tools **360**.

The graphical editors **310**, **320**, **330**, **340** are preferably fourth generation languages (4GL) or Computer Aided Software Engineering (CASE) tools that facilitate the application development task by enabling a certain amount of the application code to be generated automatically from graphical representations. In particular, the graphical user interface editor **310** can be a commercially available user interface display platforms or GUI builders discussed in the context of the presentation layer **110**.

FIG. **28** is a block diagram of the graphical user interface editor **310** of FIG. **1**. The graphical user interface editor **310** is a typical user interface made to create menus and paint screens. As such, the graphical user interface editor **310** includes a screen editor **311** to position graphical representations of business objects on a screen or form. Screens can thus include text fields, labels, buttons, selection boxes, pull down lists, and similar graphical objects that compose a user interface. These graphical representations of business objects can be grouped so that a screen can be composed of sub-screens. This is useful to represent screen overlays, which are screens that have a fixed area as well as a variable area that changes depending on user actions. Sub-screens are also useful for grouping together business objects that need

to be displayed across a number of application screens. The screen editor **311** creates internal user interface representations **312** which are processed by a user interface code generator **313** into data stored in the user interface repository **152**.

FIG. **29** is a block diagram of the graphical business process editor **320** of FIG. **1**. The graphical business process editor **320** is a tool that enables application developers to represent the business processes that an application is meant to automate in a more intuitive, graphical form, referred to as a process flowchart. Because a business process can be broken down into sub-processes, an application can be viewed as a hierarchy of process flowcharts. This modularization enables an application to look at high-level business processes separately from detailed business sub-processes. As illustrated, a business process editor **321** generates internal business process representations **322**, which are converted by a business process code generator **323** into data stored in the business process repository **154**.

In a preferred embodiment of the present invention, process flowcharts can be viewed as conventional transition diagrams. Transition diagrams are networks of nodes represented graphically by circles and are called states. The states are connected by directed labeled arrows, called edges. Edge labels represent the transformations that lead from one state to the next.

As an example, the process of applying for a driver's license includes routing a driver's license application paper form through the various departments in charge of eye exams, written test, driving test and the like, with a progression from department to department. This process can be modeled using a transition diagram in which the states represent the various departments, and the edges represent the changes to the electronic driver's license form that need to be performed before that form is routed to the next department.

A transition diagram can be deterministic, or non-deterministic, where non-deterministic means that more than one edge with the same label is possible out of a state. In a preferred embodiment of the present invention, non-deterministic transition diagrams are used to represent business processes graphically. These non-deterministic transition diagrams constitute a process representation perfectly suited for interpretation by business process engine **124** (FIG. **8**), which, as mentioned earlier, is the event handler based on NFA theory that processes the business process flowcharts resulting from the graphical business process editor **320**.

FIG. **30** is a block diagram of the graphical business object editor **330** of FIG. **1**. The graphical business object editor **330** is a tool that enables the graphical editing of business objects and their relationships. A business object editor **331** generates internal business object representations which are converted by a business object code generator into data stored in the business object repository **156**.

In a preferred embodiment of the present invention, the graphical business object editor **330** can be viewed as a Entity-Relationship (ER) diagramming tool. The ER data model is the predominant conceptual level description tool and is used as a diagramming technique where rectangles represent entities, circles represent attributes, diamonds represent relationships. This graphical ER diagram can be used to generated automatically the application database schema, and a number of the basic data accessor queries.

In addition, default constraints can be automatically associated to business objects based on the business object type. This can lead to automatic generation of maintenance screens for lookup business objects that can take a known range of values. The graphical business object editor can also be used to create templates that can be reused throughout an application. For instance, every screen may have a number of fixed function keys or buttons such as display, insert, delete, update, clear, refresh, backup, or quit, as well as a number of variable function keys whose semantics change from screen to screen. These function keys can be treated as a group and provided automatically as part of the template for every screen in an application.

FIG. **31** is a block diagram of the graphical data record editor **340** of FIG. **1**. The graphical data record editor **340** is a tool that provides access to the data stored in the relational tables of the data layer RDBMS. As such, it is an interface that provides graphical access to each application table and permits the application developer or maintainer to view, insert, delete or update specific data records. As illustrated, a data record editor **341** generates internal data record representations **342** which are converted by a data record code generator **343** into data stored in the data record repository **158**.

This focus on application data is complemented by an ability to manipulate DDL structures. In this function, the data record editor can be viewed as a data repository used to generate the database schema, either initially in its totality, or subsequently, for incremental updates. In this regards, the data record editor is similar to commercially-available off-the-shelve packages such as PeopleSoft Inc.'s Record Editor or ERwin from Logic Works Corporation. As a whole, the data record editor greatly facilitates application maintenance and data error recovery for day to day application development, maintenance, and operation.

FIG. **32** is a block diagram of the logic development environment **350** of FIG. **1**. The logic development environment **350** is an environment that adheres to the "open system" standards. As defined herein, an open system is a system that implements sufficient open specifications for interfaces, services, and supporting formats to enable properly engineered application software to be ported across a wide range of systems with minimal changes, to interoperate with other applications on local and remote systems, and to interact with users in a style which facilitates user portability. Open specifications are defined herein as a public specification that is maintained by an open, public consensus process to accommodate new technology over time and that is consistent with standards.

Functionality, the logic development environment **350** includes, at a minimum, an operating system **351**, a third generation programming language compiler **352** and debugger **353**, a runtime building facility **354**, a source control system **355**, and a screen oriented text editor **356**. One possible embodiment of the logic development environment could use the UNIX operating system, the ANSI C programming language, the XDB debugging facility, the MAKE build utility, the RCS revision control system, and the EMACS text editor.

FIG. **33** is a schematic block diagram of the facilitation tools **360** of FIG. **1**. The facilitation tools **360** are graphic editing tools. The primary concept is to provide a structured, yet flexible, methodology for gathering user and application requirements while enabling the use of the resulting documentation to automatically generate a number of the architectural constructs that would otherwise have to be encoded manually. These facilitation tools **360** can include project tools **361**, organizational tools **362**, communication tools

363, office tools 364, groupware tools 365, and templates 366 for processing user inputs.

Equivalents

While this invention has been particularly shown and described with reference to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention as defined by the appended claims. In particular, the invention is not limited to particular communications links, protocols, data structure formats, etc. In addition, although various features of the invention are disclosed as being either hardware or software, it is understood that any feature of the invention can be embodied in hardware, software or firmware.

These and all other equivalents are intended to be encompassed by the following claims.

What is claimed is:

1. A system to transition legacy applications operable on a legacy computing system to a distributed infrastructure, the system comprising:

a multi-tiered computer architecture including a process control tier for modeling an internal work flow of an enterprise and a functionality tier for performing work in accordance with the work flow of the enterprise; and

a legacy application inoperable on the multi-tiered computer architecture and

an automated converter to transition the legacy application to a target application operable on the multi-tiered computer architecture.

2. The system of claim 1 wherein the multi-tiered computer architecture is a client-server architecture having at least four tiers.

3. The system of claim 1 where the multi-tiered architecture further includes a presentation tier for interfacing with a user, a data retrieval tier, and a data storage tier.

4. The system of claim 1 wherein the converter includes an intermediate language, the language of the legacy application being translated to the intermediate language and from the intermediate language to the language of the target application.

5. The system of claim 1 wherein the target application is an internet accessible application.

6. The system of claim 1 wherein the target application is an object-oriented application.

7. The system of claim 1 further comprising an implementation plan which provides a prioritized list of legacy applications to be transitioned by the automated converter.

8. The system of claim 7 wherein the implementation plan further provides instructions for controlling the operation of the automated converter.

9. The system of claim 1 further comprising an implementation strategy which identifies inputs to a target system and provides a list of action items for obtaining the identified outputs from the legacy computing system.

10. A method for transitioning legacy applications operable on a legacy computing system to a distributed infrastructure, comprising the steps of:

providing a multi-tiered architecture including a process control tier for modeling an internal work flow of an enterprise and a functionality tier for performing work in accordance with the work flow of the enterprise; and

providing a legacy application inoperable on the multi-tiered computer architecture; and

in a computer, automatically converting the legacy application to a target application operable on the multi-tiered computer architecture.

11. The method of claim 10 wherein the step of providing a multi-tiered architecture comprises providing a client-server architecture having at least four tiers.

12. The method of claim 10 where the multi-tiered computer architecture further includes a presentation tier for interfacing with a user, a data retrieval tier, and a data storage tier.

13. The method of claim 10 wherein the step of converting comprises translating the language of the legacy application to an intermediate language and translating the intermediate language to the language of the target application.

14. The method of claim 10 wherein the target application is an internet accessible application.

15. The method of claim 10 wherein the target application is an object-oriented application.

16. The method of claim 10 further comprising the step of providing a prioritized list of legacy applications to be transitioned by the automated converter.

17. The method of claim 16 further comprising the step of controlling the operation of the automated converter based on the prioritized list.

18. The method of claim 10 further comprising the steps of:

identifying inputs to a target system including the target application; and

obtaining the identified outputs from the legacy applications from a list of action items.

19. In a computer, a converter for translating a legacy program component operating in a legacy language on a legacy computing system to a target program component operating in a target language on a distributed infrastructure includes a process control tier and a functionality tier, the converter comprising:

an intermediate language;

a first converter for translating the legacy program component from the legacy language to an intermediate component in the intermediate language wherein the legacy language is operable on a legacy processor; and

a second converter for translating the intermediate component from the intermediate language to the target program component in the target language wherein the target language is operable on a target processor different from the legacy processor.

20. The converter of claim 19 wherein the intermediate language is independent of the legacy language and the target language.

21. The converter of claim 19 wherein the intermediate language is inoperable on either the legacy processor or the target processor.

22. The converter of claim 19 wherein the first converter parses the legacy program component into a parse tree, the intermediate component representing the parse tree in a postfix notation.

23. In a computer, a method for translating a legacy program component operating in a legacy language on a legacy computing system to a target program component operating in a target language on a distributed infrastructure includes a process control tier and a functionality tier, comprising the steps of:

defining an intermediate language;

in a first converter, translating the legacy program component from the legacy language to an intermediate component in the intermediate language wherein the legacy language is operable on a legacy processor; and

in a second converter, translating the intermediate component from the intermediate language to the target

program in the target language wherein the target language is operable on a target processor different from the legacy processor.

24. The method of claim 23 wherein the intermediate language independent of the legacy language and the target language.

25. The method of claim 23 wherein the intermediate language is inoperable on either the legacy processor or the target processor.

26. The method of claim 23 wherein the step of translating the legacy program component comprises:

parsing the legacy program component into a parse tree; and

representing the parse tree in a postfix notation.

27. A system to transition legacy applications operable on a legacy computing system to a distributed infrastructure, the system comprising:

a multi-tiered computer architecture including a process control tier for modeling an internal work flow of an enterprise and a functionality tier for performing work in accordance with the work flow of the enterprise; and

an automated converter having a first converter and a second converter to transition a legacy application to a target application operable on the multi-tiered computer architecture, the first converter translating the legacy application from a legacy language to an intermediate language and the second converter translating the translated legacy application from the intermediate language to the target application in a target language.

28. The system of claim 27 wherein the multi-tiered computer architecture is a client-server architecture having at least four tiers.

29. The system of claim 27 where the multi-tiered computer architecture further includes a presentation tier for interfacing with a user, a data retrieval tier, and a data storage tier.

30. The system of claim 27 wherein the intermediate language is independent of the legacy language and the target language.

31. The system of claim 27 wherein the target application is an internet accessible application.

32. The system of claim 27 wherein the target application is an object-oriented application.

33. The system of claim 27 further comprising an implementation plan which provides a prioritized list of legacy applications to be transitioned by the automated converter.

34. The system of claim 33 wherein the implementation plan further provides instructions for controlling the operation of the automated converter.

35. The system of claim 27 further comprising an implementation strategy which identifies inputs to a target system including the target applications and provides a list of action items for obtaining the identified outputs from the legacy applications.

36. A method for transitioning legacy applications operable on a legacy computing system to a distributed infrastructure, comprising the steps of:

providing a multi-tiered computer architecture including a process control tier for modeling an internal work flow of an enterprise and a functionality tier for performing work in accordance with the work flow of the enterprise; and

in a computer, automatically converting a legacy application to a target application operable on the multi-tiered computer architecture, the legacy application translated from a legacy language to an intermediate language and the translated legacy application being translated from the intermediate language to the target application in a target language.

37. The method of claim 36 wherein the step of providing comprises providing a client-server architecture having at least four tiers.

38. The method of claim 36 where the multi-tiered computer architecture further includes a presentation tier for interfacing with a user, a data retrieval tier, and a data storage tier.

39. The method of claim 36 wherein the step of converting comprises defining the intermediate language to be independent of the legacy language and the target language.

40. The method of claim 36 wherein the target application is an internet accessible application.

41. The method of claim 36 wherein the target application is an object-oriented application.

42. The method of claim 36 further comprising the step of prioritizing a list of legacy applications to be transitioned by the automated converter.

43. The method of claim 42 further comprising the step of controlling the operation of the automated converter based on the prioritized list.

44. The method of claim 36 further comprising the steps of:

identifying inputs to a target system including the target application; and

obtaining the identified outputs from the source applications using a list of action items.

45. A system for transitioning a legacy application operable on a legacy computing system to a distributed infrastructure includes a process control tier and a functionality tier, the system comprising:

an intermediate language;

a first converter for translating the legacy application component from a legacy language to an intermediate component in the intermediate language wherein the legacy language is operable on a legacy processor; and

a second converter for translating the intermediate component from the intermediate language to a target application component in a target language operable on the distributed infrastructure wherein the target language is operable on a target processor different from the legacy processor.

46. The system of claim 45 wherein the intermediate language is independent of the legacy language and the target language.

47. The system of claim 45 wherein the intermediate language is inoperable on either the legacy processor or the target processor.

48. The system of claim 45 wherein the first converter parses the legacy program component into a parse tree, the intermediate component representing the parse tree in a postfix notation.

49. The system of claim 45 further comprising a multi-tiered computer architecture.

50. A method for transitioning a legacy application operable on a legacy computing system to a distributed infrastructure includes a process control tier and a functionality tier, comprising the steps of:

defining an intermediate language;

in a first converter, translating the legacy application component from a legacy language to an intermediate component in the intermediate language wherein the legacy language is operable on a legacy processor; and

in a second converter, translating the intermediate component from the intermediate language to a target

application component in a target language operable on the distributed infrastructure wherein the target language is operable on a target processor different from the legacy processor.

51. The method of claim **50** wherein the intermediate language independent of the legacy language and the target language.

52. The method of claim **50** wherein the intermediate language is inoperable on either the legacy processor or the target processor.

53. The method of claim **50** wherein the step of translating the legacy program component comprises:

parsing the legacy program component into a parse tree; and

representing the parse tree in a postfix notation.

54. The method of claim **50** further comprising the step of operating the target program on a multi-tiered computer architecture.

* * * * *

US006269396B1

(12) **United States Patent**
Shah et al.

(10) **Patent No.: US 6,269,396 B1**
(45) **Date of Patent: Jul. 31, 2001**

(54) **METHOD AND PLATFORM FOR INTERFACING BETWEEN APPLICATION PROGRAMS PERFORMING TELECOMMUNICATIONS FUNCTIONS AND AN OPERATING SYSTEM**

(75) Inventors: **Mahesh V. Shah**, Plano; **David W. McDaniel**, Dallas; **James R. Vatteroni**, Allen; **Stephen B. Jaggers**, Allen; **Mark E. Worline**, Allen, all of TX (US)

(73) Assignee: **Alcatel USA Sourcing, L.P.**, Plano, TX (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/211,016**

(22) Filed: **Dec. 11, 1998**

Related U.S. Application Data

(60) Provisional application No. 60/069,576, filed on Dec. 12, 1997.

(51) **Int. Cl.**[7] **G06F 13/00**
(52) **U.S. Cl.** **709/223**
(58) **Field of Search** 379/113, 201, 379/242; 709/200, 220, 221, 222, 223–224

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,940,487 * 8/1999 Bunch et al. 379/201

FOREIGN PATENT DOCUMENTS

93205508 10/1993 (WO) G06F/9/40
9707638 2/1997 (WO) H04Q/3/00
9724837 7/1997 (WO) H04L/12/24
9731451 8/1997 (WO) H04L/12/24

OTHER PUBLICATIONS

Maltini, et al. "OSI System and Application Management: an Experience in a Public Administration Context", pp. 492–500, Apr. 492–500, Apr. 15, 1996, IEEE.

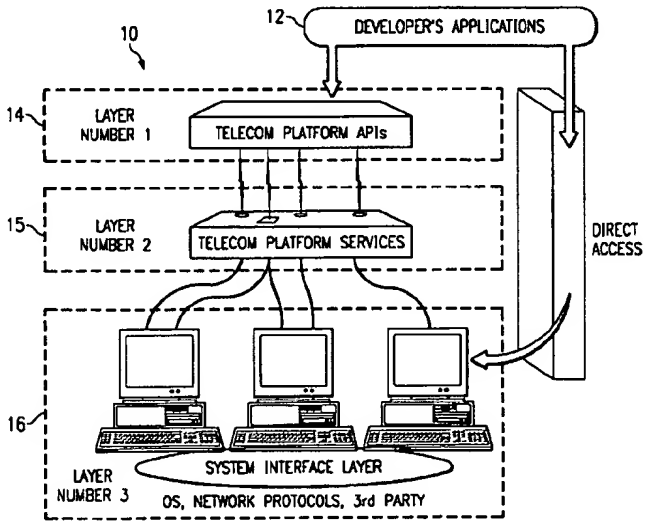
* cited by examiner

Primary Examiner—Robert B. Harrell
(74) *Attorney, Agent, or Firm*—Baker Botts L.L.P.

(57) **ABSTRACT**

A method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network is provided. The method includes providing a network platform manager operable to remove nodes from service, restore nodes to service, remove applications from service, and restore applications to service, providing a network system integrity manager operable to monitor the nodes and to enable failed nodes to recover, providing a configuration manager operable to interface with a host coupled to the telecom platform, providing a node platform manager operable to provide management functions for a node, providing a service manager operable to start and stop processes at the direction of the node platform manager, and providing a node system integrity manager operable to monitor inter-node links.

42 Claims, 21 Drawing Sheets



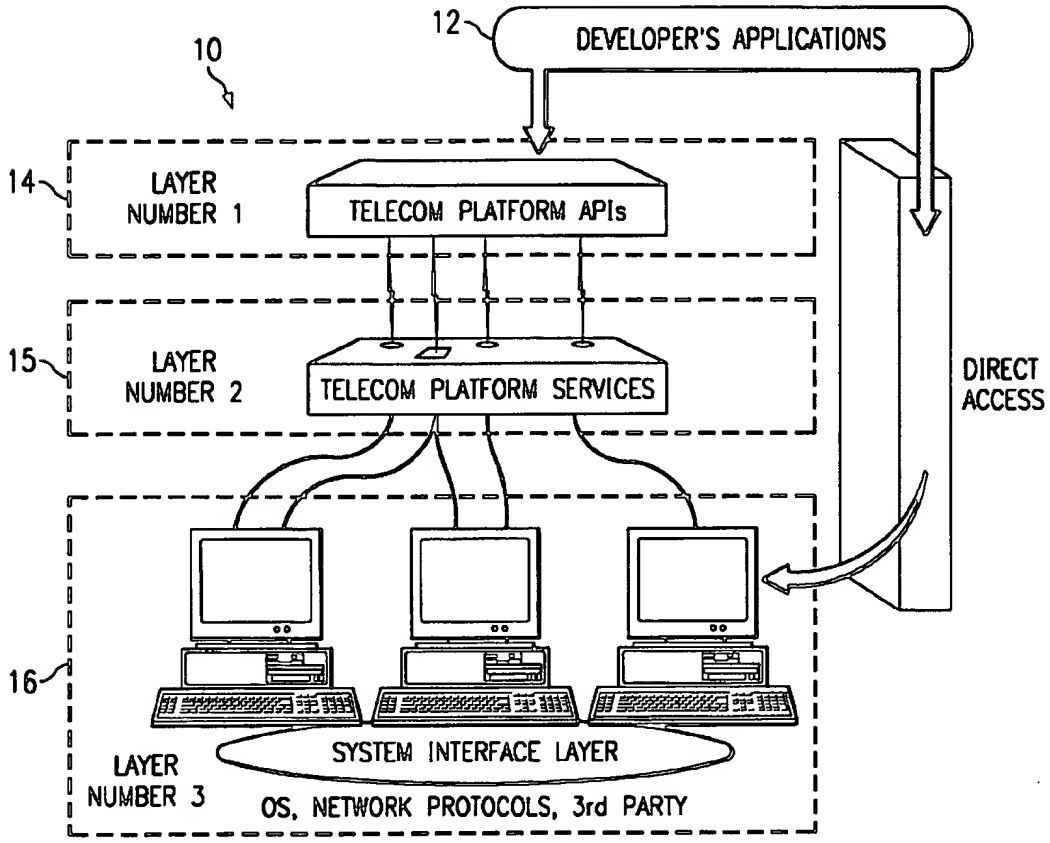

12
DEVELOPER'S APPLICATIONS
10
14
LAYER
NUMBER 1
TELECOM PLATFORM APIs
15
LAYER
NUMBER 2
TELECOM PLATFORM SERVICES
DIRECT
ACCESS
16
SYSTEM INTERFACE LAYER
LAYER
NUMBER 3
OS, NETWORK PROTOCOLS, 3rd PARTY

FIG. 1

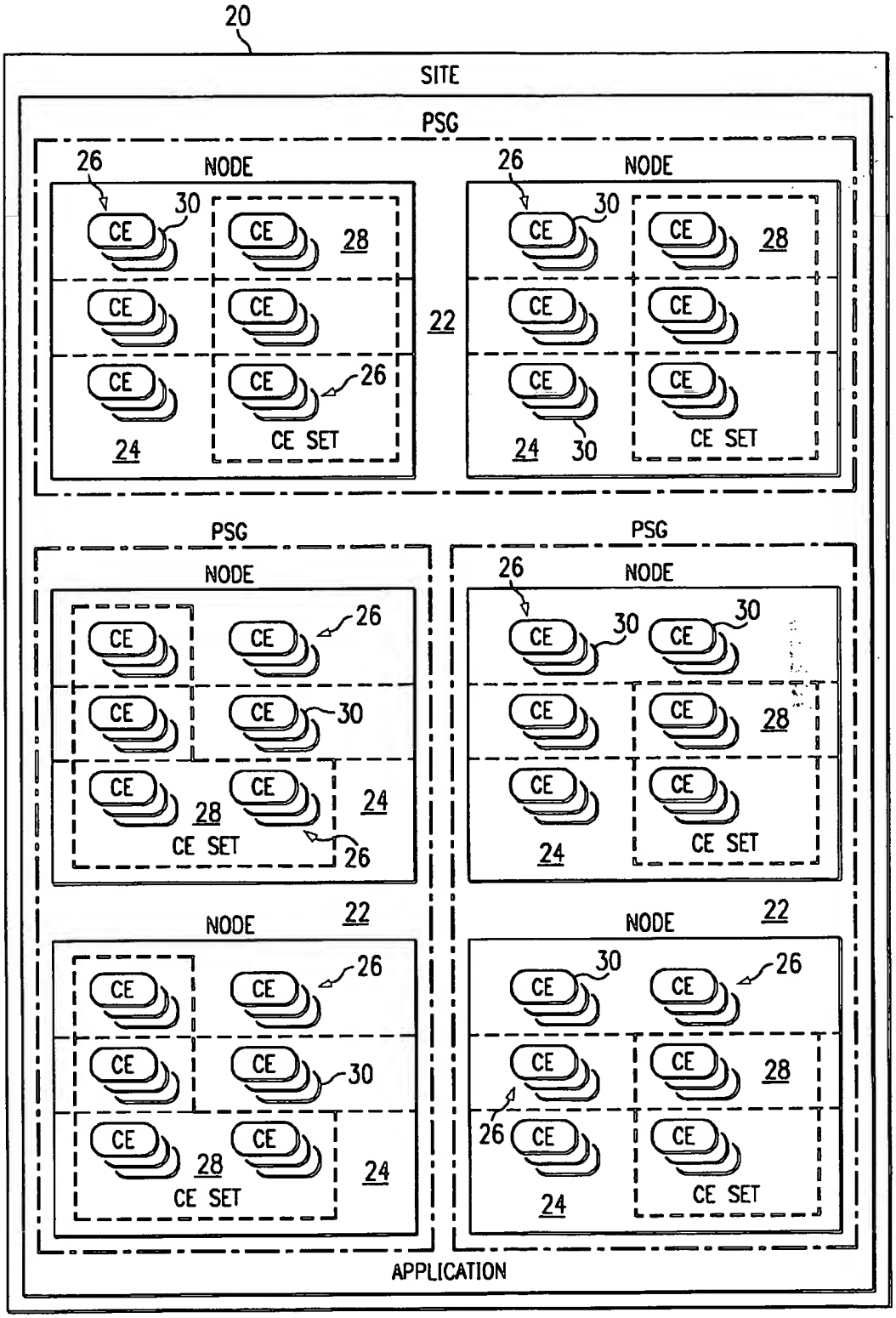


FIG. 2

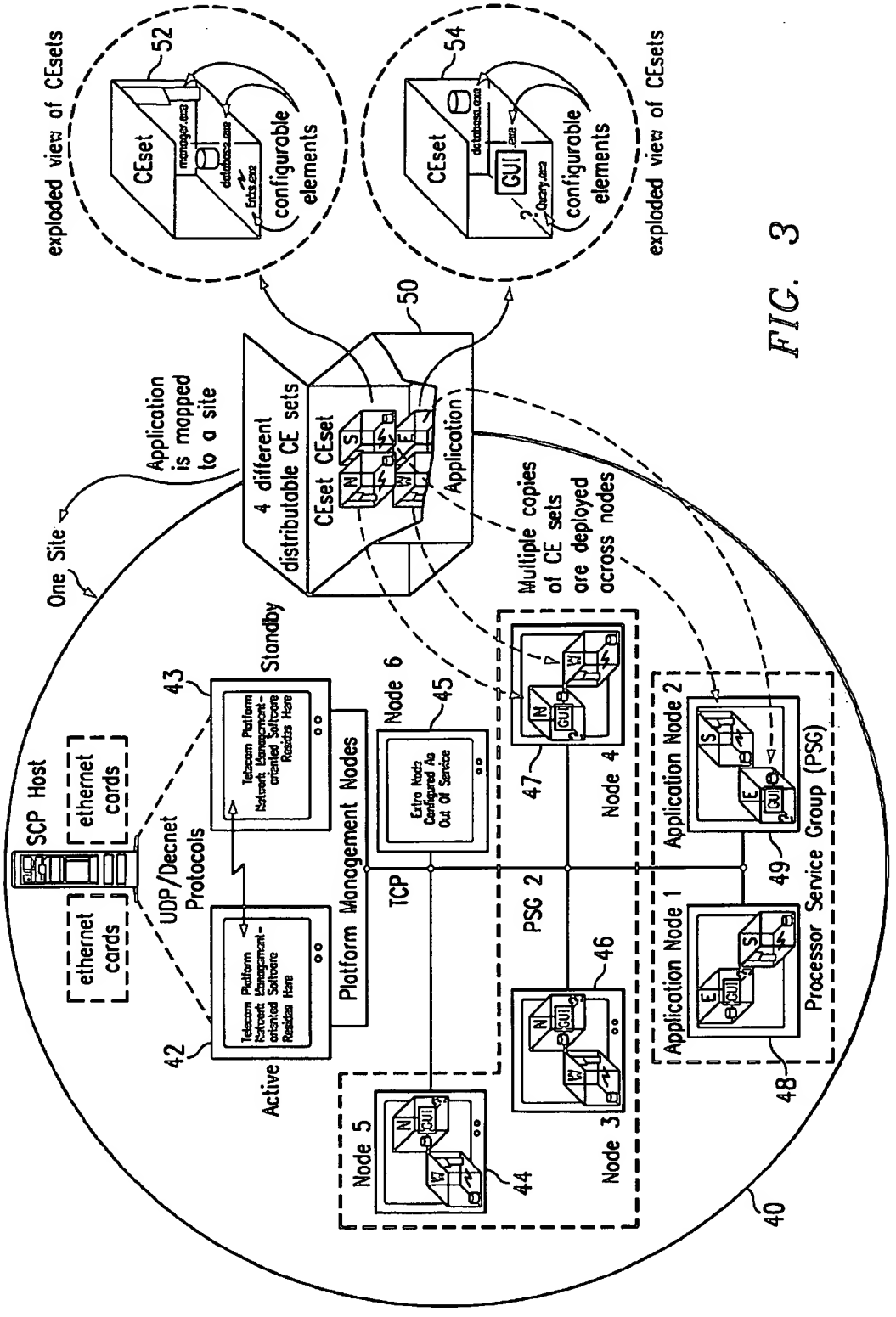

exploded view of CEsets
52
CEset
configurable elements
54
CEset
configurable elements
exploded view of CEsets
50
4 different distributable CE sets
CEset CEset
Application
Application is mapped to a site
One Site
Multiple copies of CE sets are deployed across nodes
Standby
43
Telecom Platform Network Management-oriented Software Resides Here
Node 6
45
Extra Node Configured As Out of Service
SCP Host
ethernet cards
ethernet cards
UDP/Decnet Protocols
Platform Management Nodes
TCP
PSC 2
47
Node 4
Application Node 2
49
Application Node 1
Processor Service Group (PSG)
48
42
Active
46
Node 5
44
Node 3
40

FIG. 3

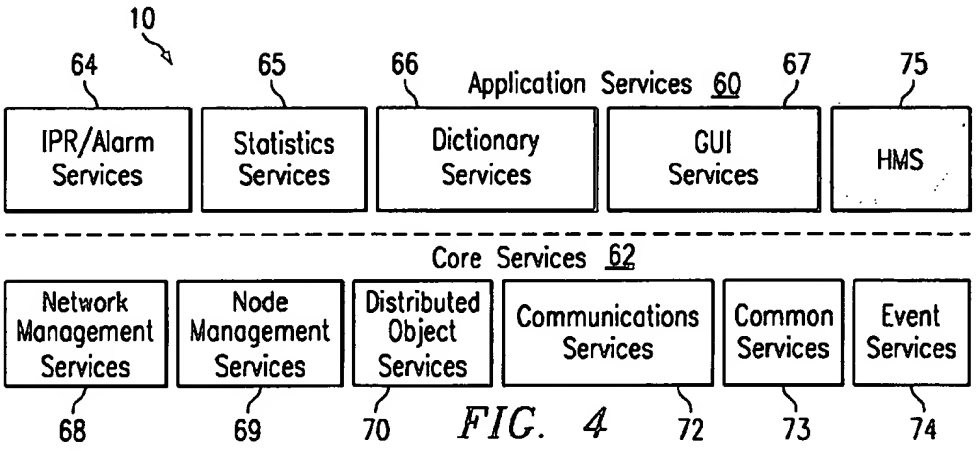

10
64
65
66
Application Services 60
67
75
IPR/Alarm Services
Statistics Services
Dictionary Services
GUI Services
HMS
Core Services 62
Network Management Services
Node Management Services
Distributed Object Services
Communications Services
Common Services
Event Services
68
69
70
72
73
74

FIG. 4

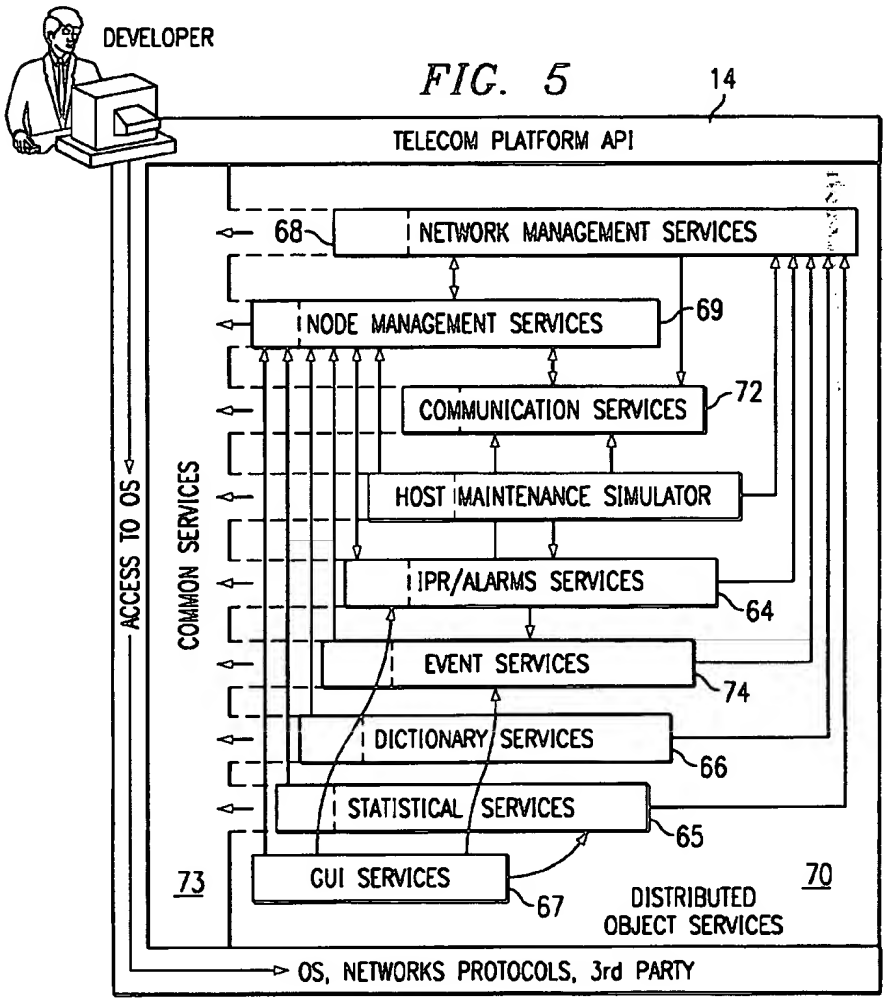

DEVELOPER
14
TELECOM PLATFORM API
68
NETWORK MANAGEMENT SERVICES
NODE MANAGEMENT SERVICES
69
COMMUNICATION SERVICES
72
HOST MAINTENANCE SIMULATOR
IPR/ALARMS SERVICES
64
EVENT SERVICES
74
DICTIONARY SERVICES
66
STATISTICAL SERVICES
65
GUI SERVICES
67
73
70
DISTRIBUTED OBJECT SERVICES
ACCESS TO OS
COMMON SERVICES
OS, NETWORKS PROTOCOLS, 3rd PARTY

FIG. 5

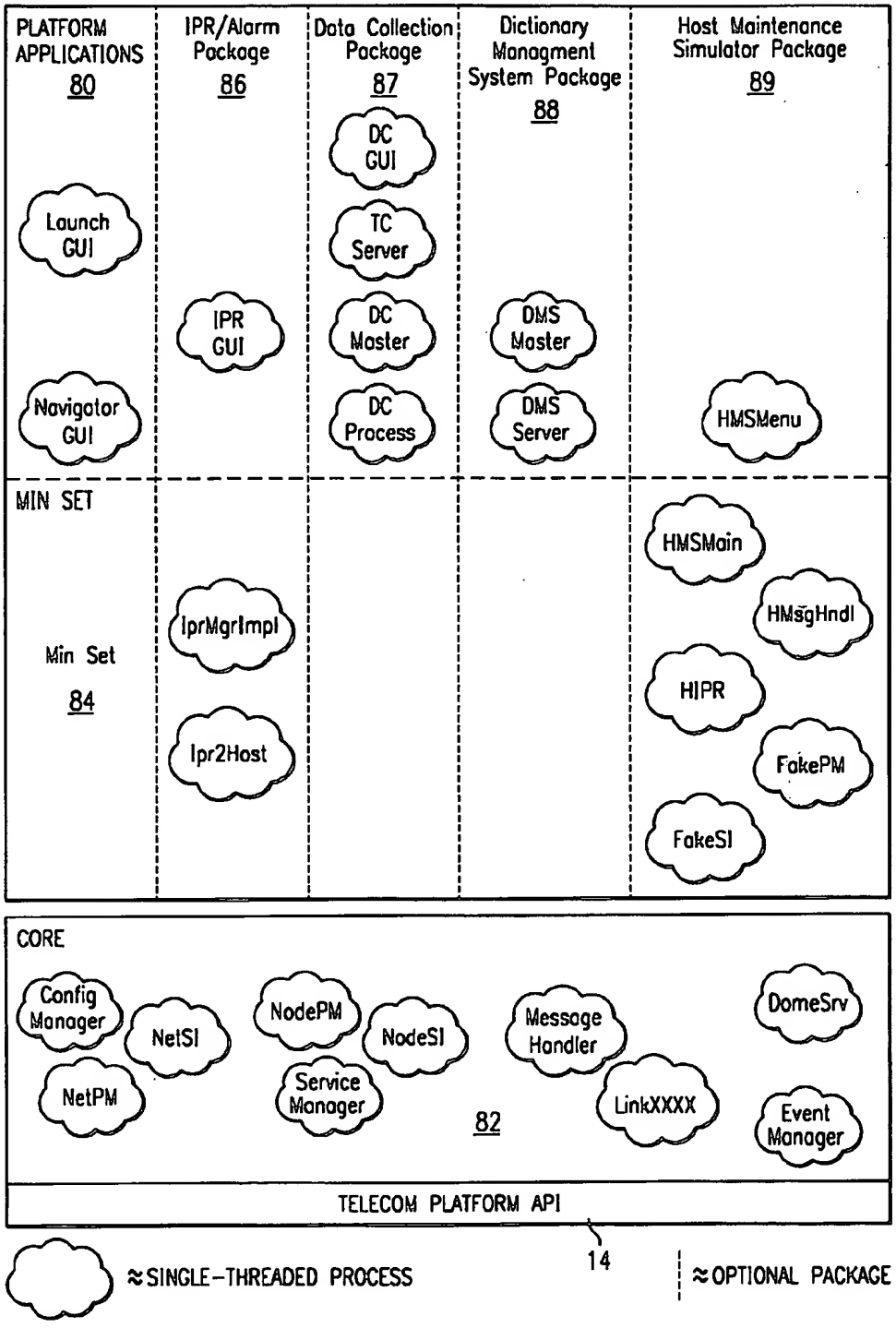

PLATFORM APPLICATIONS
80
IPR/Alarm Package
86
Data Collection Package
87
Dictionary Managment System Package
88
Host Maintenance Simulator Package
89
DC GUI
Launch GUI
TC Server
IPR GUI
DC Master
DMS Master
Navigator GUI
DC Process
DMS Server
HMSMenu
MIN SET
HMSMain
HMsgHndl
IprMgrImpl
Min Set
84
HIPR
Ipr2Host
FakePM
FakeSI
CORE
Config Manager
NetSI
NodePM
NodeSI
Message Handler
DomeSrv
NetPM
Service Manager
LinkXXXX
Event Manager
82
TELECOM PLATFORM API
14
≈SINGLE-THREADED PROCESS
≈OPTIONAL PACKAGE

FIG. 6

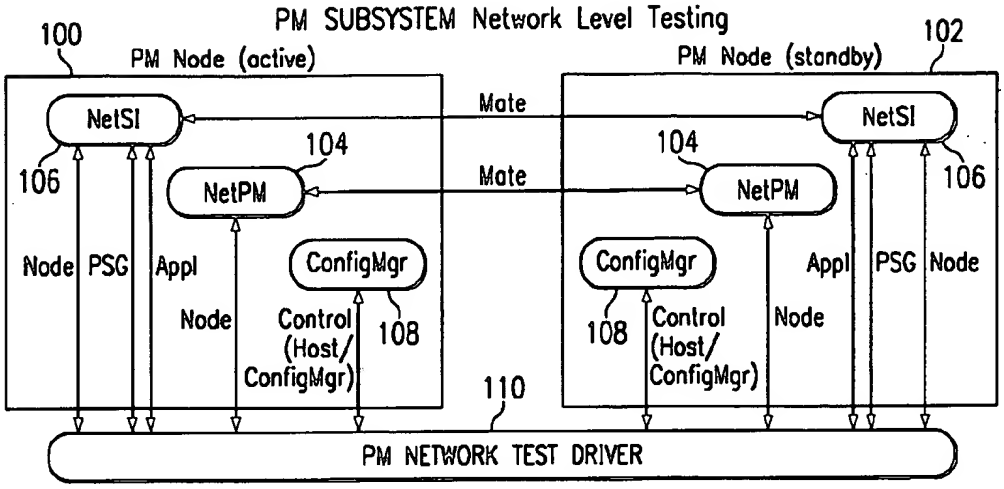

PM SUBSYSTEM Network Level Testing
100
PM Node (active)
102
PM Node (standby)
NetSI
Mate
NetSI
104
104
106
NetPM
Mate
NetPM
106
Node
PSG
Appl
ConfigMgr
ConfigMgr
Appl
PSG
Node
Node
Control (Host/ ConfigMgr)
108
108
Control (Host/ ConfigMgr)
Node
110
PM NETWORK TEST DRIVER

FIG. 7A

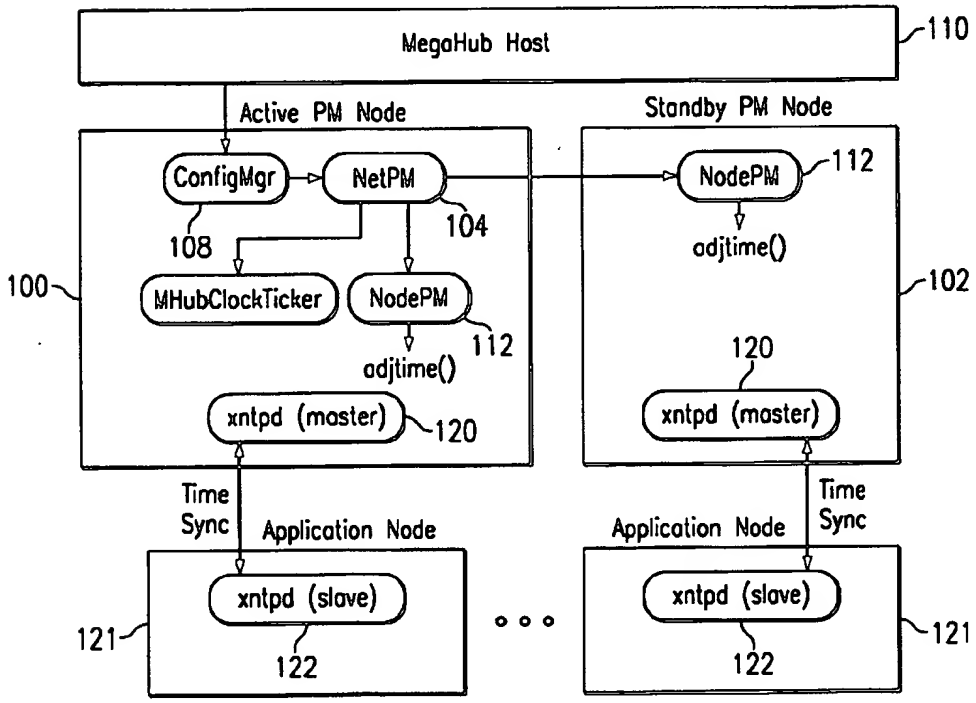

MegaHub Host
110
Active PM Node
Standby PM Node
ConfigMgr
NetPM
NodePM
112
108
104
adjtime()
100
MHubClockTicker
NodePM
102
112
adjtime()
120
xntpd (master)
120
xntpd (master)
Time Sync
Application Node
Application Node
Time Sync
xntpd (slave)
xntpd (slave)
121
122
122
121

FIG. 7B

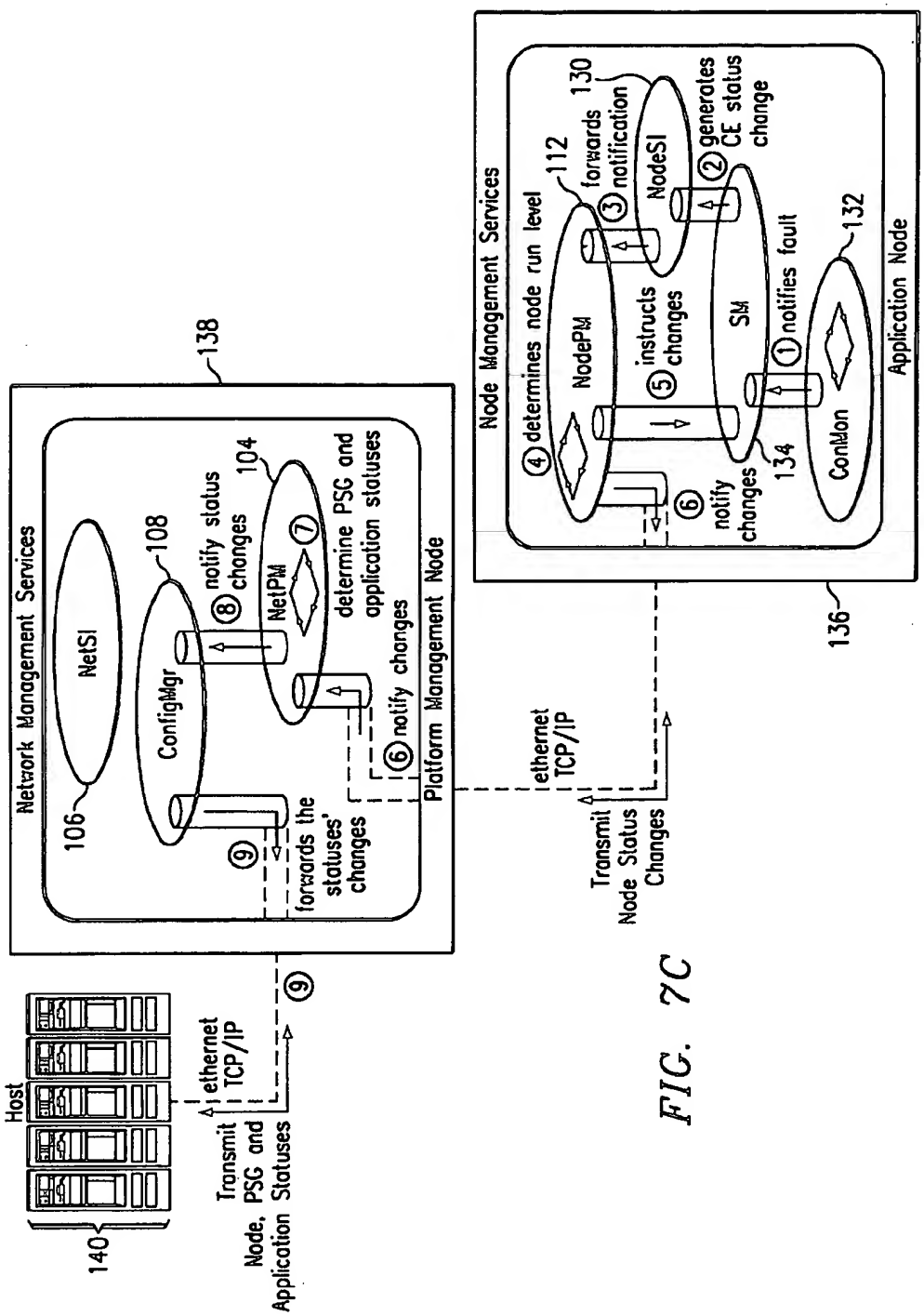

Node Management Services
Application Node
Platform Management Node
Network Management Services
NodePM
NodeSI
SM
ConMon
NetSI
ConfigMgr
NetPM
Host
ethernet TCP/IP
Transmit Node Status Changes
Transmit PSG and Application Statuses
① notifies fault
② generates CE status change
③ notification forwards
④ determines node run level
⑤ instructs changes
⑥ notify changes
⑦ determine PSG and application statuses
⑧ notify status changes
⑨ forwards the statuses' changes
106
108
104
112
130
132
134
136
138
140

FIG. 7C

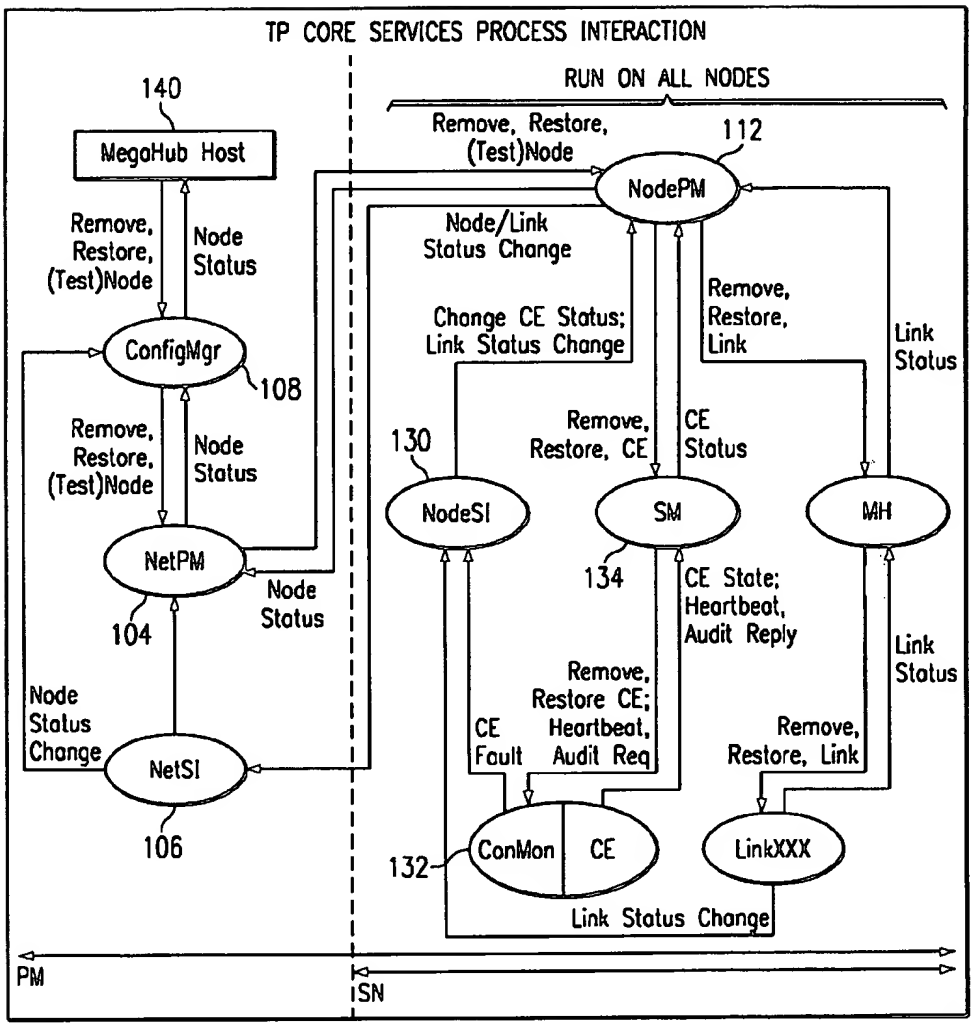

TP CORE SERVICES PROCESS INTERACTION
RUN ON ALL NODES
140
MegaHub Host
Remove, Restore, (Test)Node
Node Status
ConfigMgr
108
NetPM
104
NetSI
106
Node Status Change
112
NodePM
Remove, Restore, (Test)Node
Node/Link Status Change
Change CE Status; Link Status Change
Remove, Restore, Link
Link Status
130
NodeSI
Remove, Restore, CE
CE Status
SM
MH
134
CE State; Heartbeat, Audit Reply
Remove, Restore CE; CE Fault Heartbeat, Audit Req
Remove, Restore, Link
132
ConMon
CE
LinkXXX
Link Status Change
PM
SN

FIG. 7D

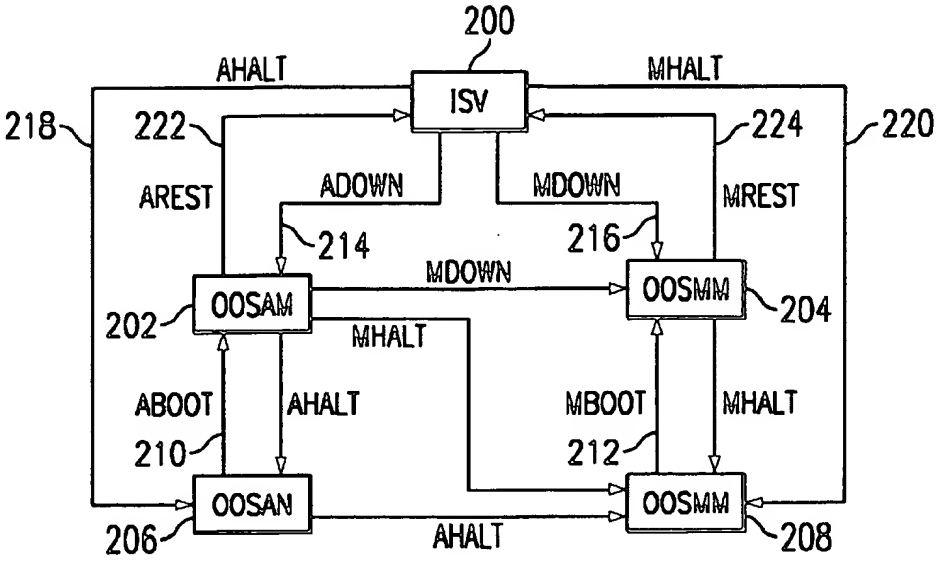

200
AHALT
MHALT
ISV
218
222
224
220
AREST
ADOWN
MDOWN
MREST
214
216
MDOWN
OOSAM
OOSMM
202
204
MHALT
ABOOT
AHALT
MBOOT
MHALT
210
212
OOSAN
OOSMM
206
208
AHALT

FIG. 8

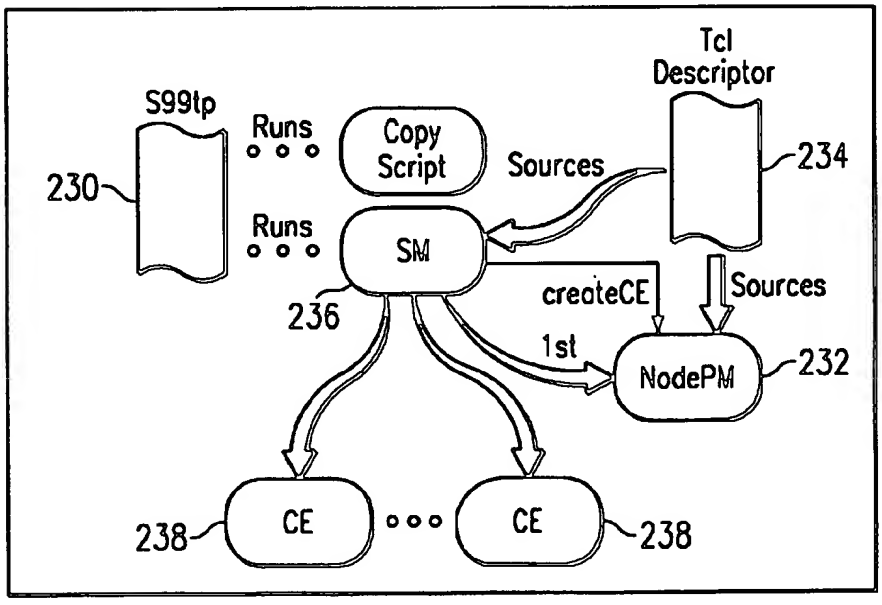

Tcl
Descriptor
S99tp
Runs
Copy
Script
Sources
234
230
Runs
SM
createCE
Sources
236
1st
NodePM
232
CE
CE
238
238

FIG. 9A

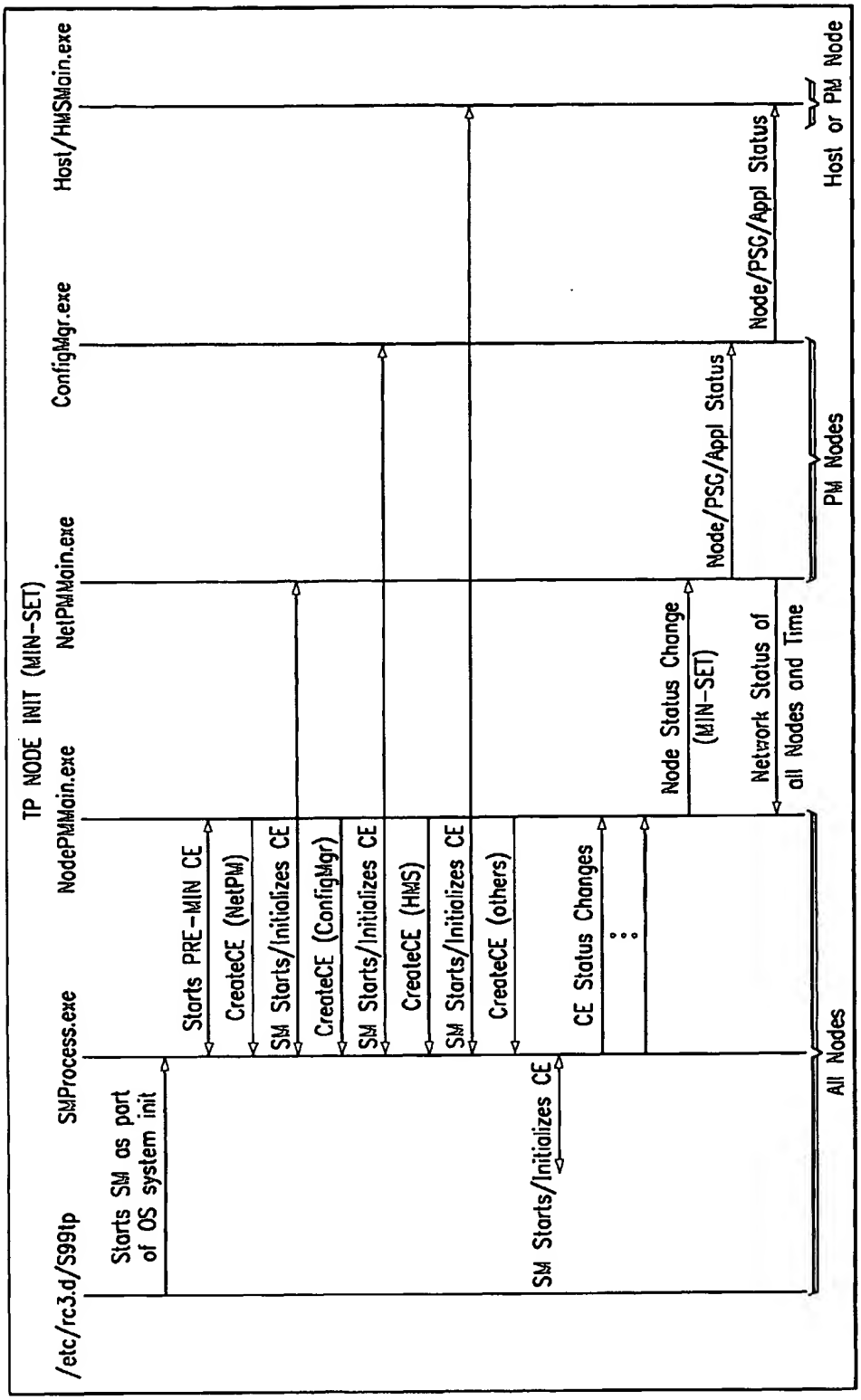

TP NODE INIT (MIN-SET)
/etc/rc3.d/S99tp
SMProcess.exe
NodePMMain.exe
NetPMMain.exe
ConfigMgr.exe
Host/HMSMain.exe
Starts SM as part of OS system init
Starts PRE-MIN CE
CreateCE (NetPM)
SM Starts/Initializes CE
CreateCE (ConfigMgr)
SM Starts/Initializes CE
CreateCE (HMS)
SM Starts/Initializes CE
CreateCE (others)
SM Starts/Initializes CE
CE Status Changes
Node Status Change (MIN-SET)
Network Status of all Nodes and Time
Node/PSG/Appl Status
Node/PSG/Appl Status
All Nodes
PM Nodes
Host or PM Node

FIG. 9B

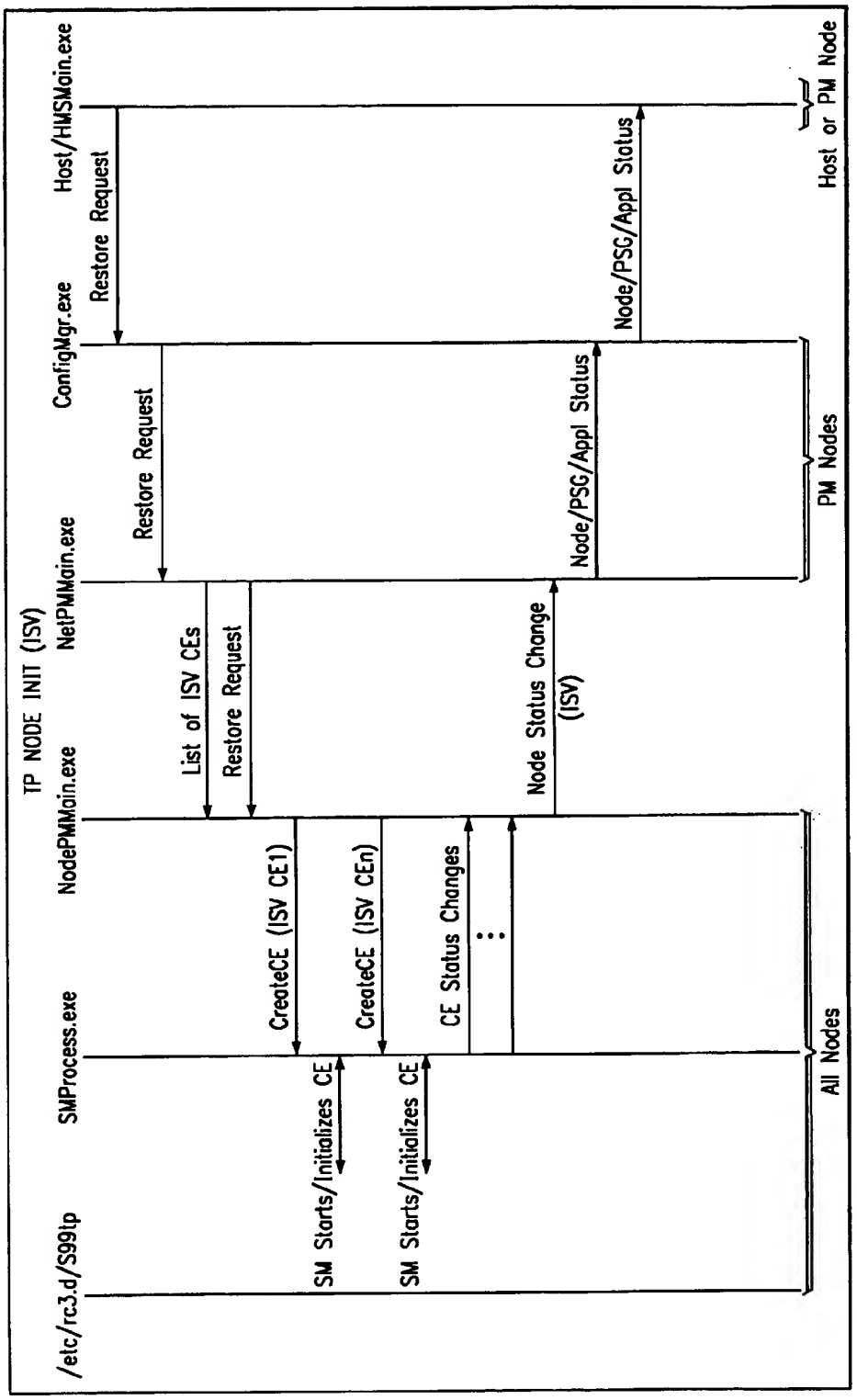

TP NODE INIT (ISV)
/etc/rc3.d/S99tp
SMProcess.exe
NodePMMain.exe
NetPMMain.exe
ConfigMgr.exe
Host/HMSMain.exe
Host or PM Node
Restore Request
Node/PSG/Appl Status
PM Nodes
List of ISV CEs
Node Status Change (ISV)
CreateCE (ISV CE1)
CreateCE (ISV CEn)
CE Status Changes
SM Starts/Initializes CE
All Nodes

FIG. 9C

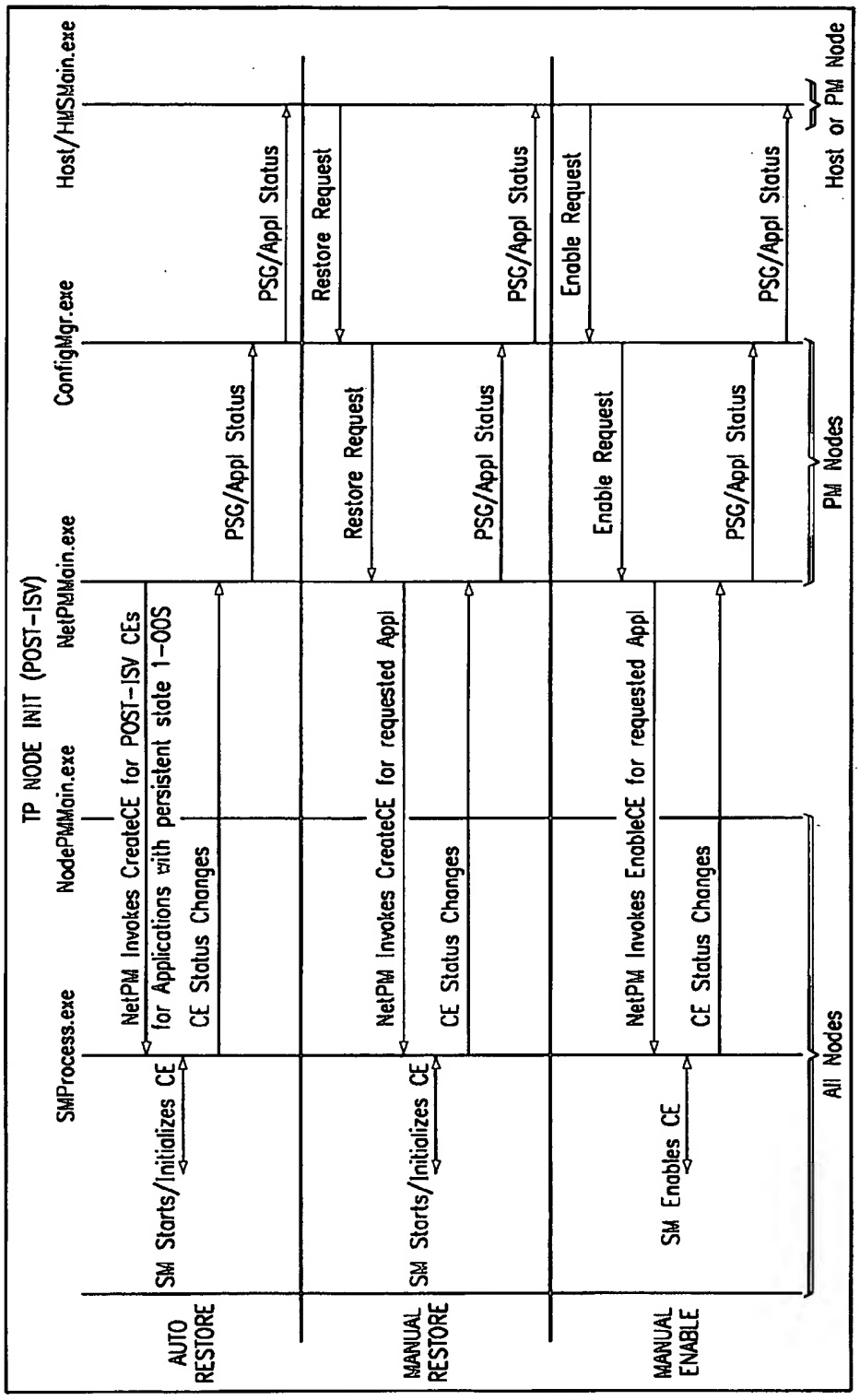
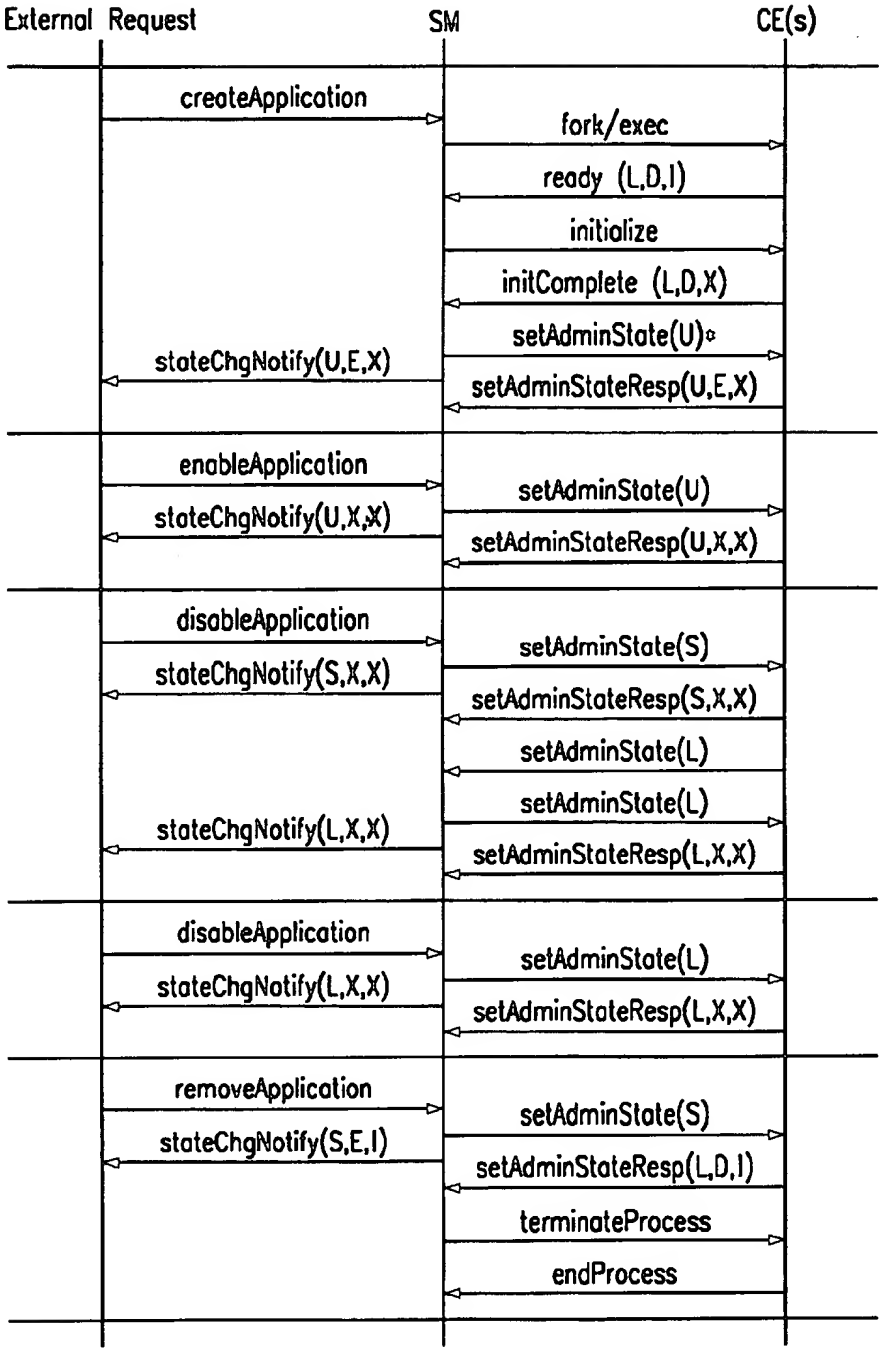

TP NODE INIT (POST-ISV)
SMProcess.exe
NodePMMain.exe
NetPMMain.exe
ConfigMgr.exe
Host/HMSMain.exe
AUTO RESTORE
MANUAL RESTORE
MANUAL ENABLE
SM Starts/Initializes CE
NetPM Invokes CreateCE for POST-ISV CEs for Applications with persistent state 1-00S
CE Status Changes
PSG/Appl Status
SM Starts/Initializes CE
NetPM Invokes CreateCE for requested Appl
CE Status Changes
Restore Request
PSG/Appl Status
SM Enables CE
NetPM Invokes EnableCE for requested Appl
CE Status Changes
Enable Request
PSG/Appl Status
All Nodes
PM Nodes
Host or PM Node

FIG. 9D

FIG. 10



I – Idle D – Disabled U – Unlocked X – Don't care
E – Enabled S – Shutdown L – Locked

* This message will only be sent if the Auto Unlocked status is set to AUTO_UNLOCKED

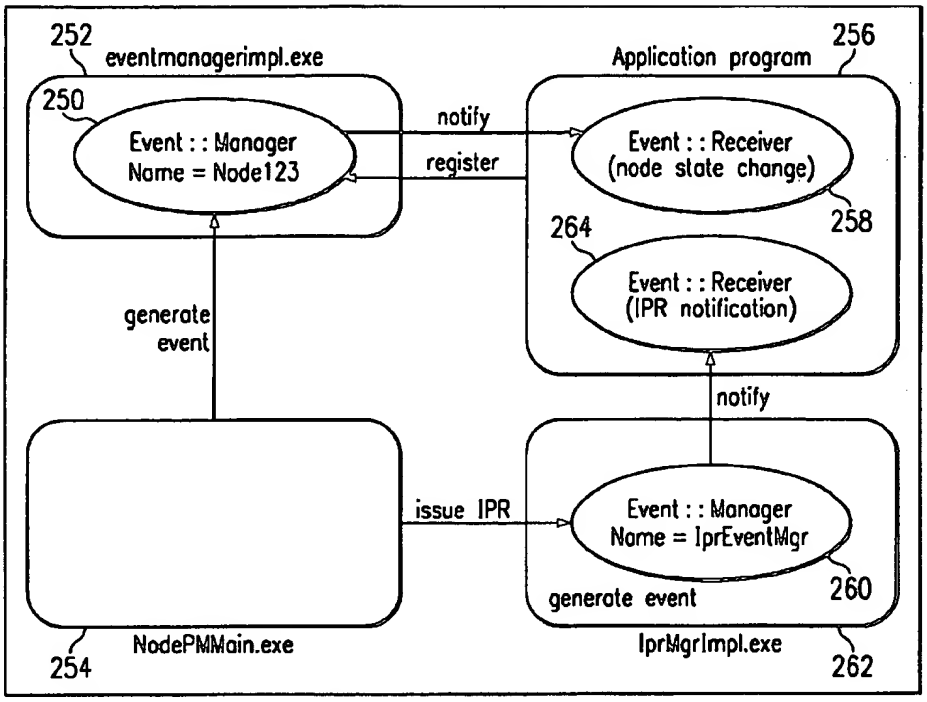
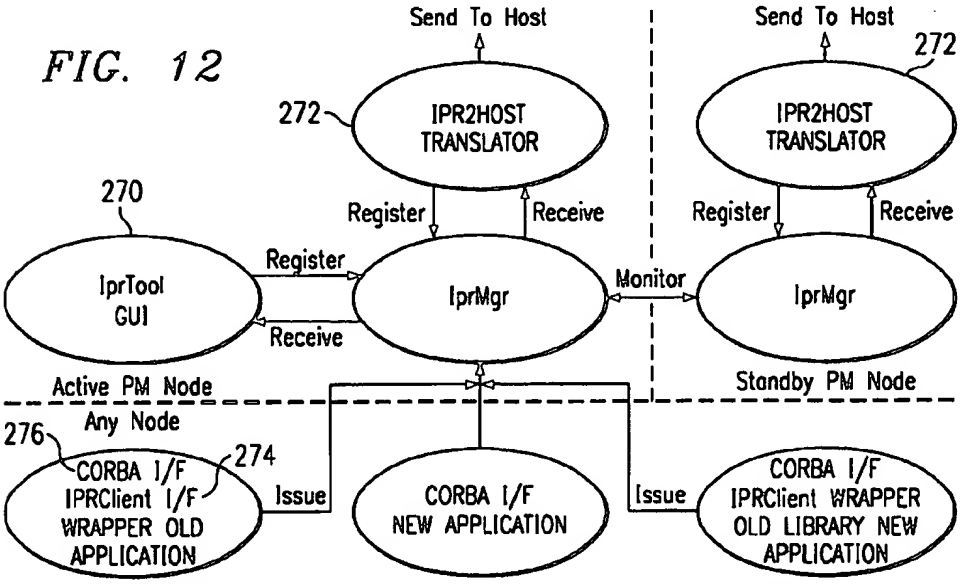


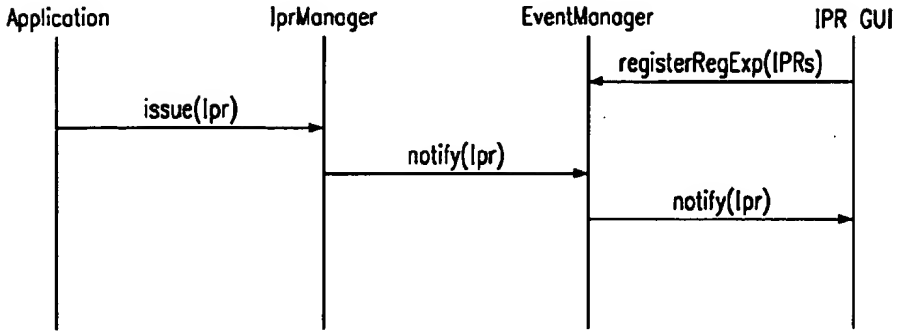
FIG. 11

FIG. 12



Client Processes Issued Requests are Forwarded to IprMgr in Active PM

FIG. 13


Application
IprManager
EventManager
IPR GUI
registerRegExp(IPRs)
issue(Ipr)
notify(Ipr)
notify(Ipr)

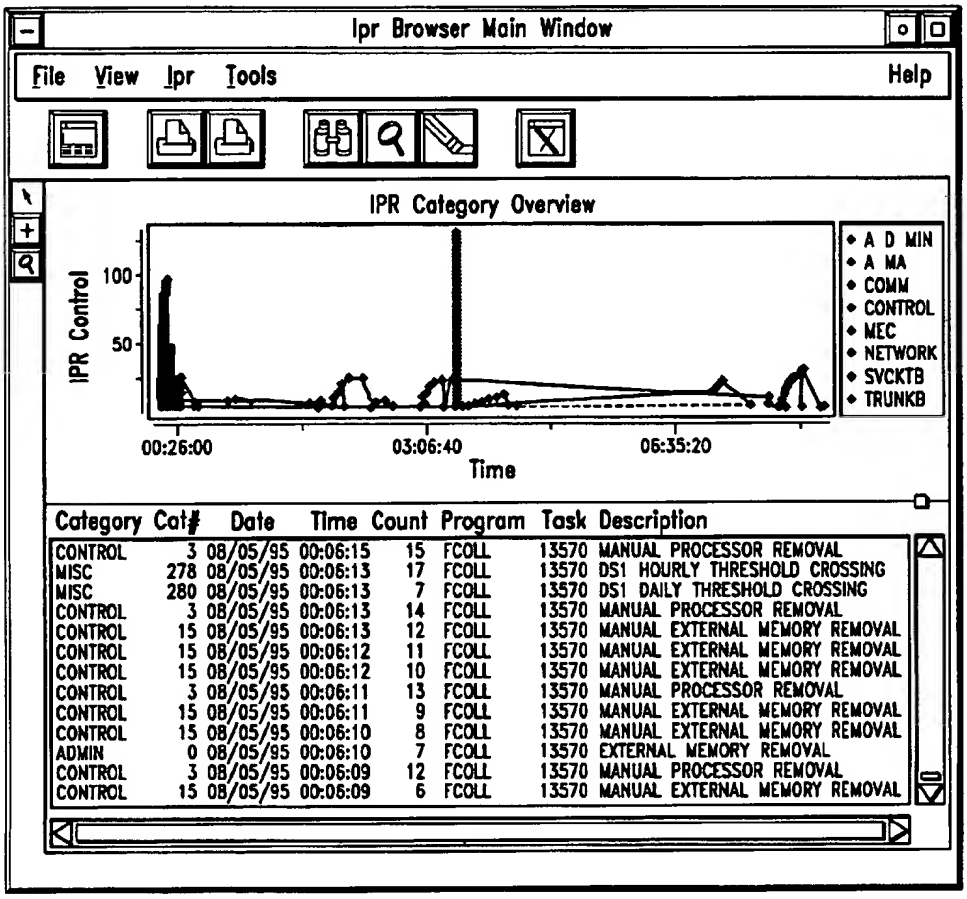

Ipr Browser Main Window
File View Ipr Tools Help
IPR Category Overview
IPR Control
100
50
00:26:00
03:06:40
06:35:20
Time
A D MIN
A MA
COMM
CONTROL
MEC
NETWORK
SVCKTB
TRUNKB
Category Cat# Date Time Count Program Task Description
CONTROL 3 08/05/95 00:06:15 15 FCOLL 13570 MANUAL PROCESSOR REMOVAL
MISC 278 08/05/95 00:06:13 17 FCOLL 13570 DS1 HOURLY THRESHOLD CROSSING
MISC 280 08/05/95 00:06:13 7 FCOLL 13570 DS1 DAILY THRESHOLD CROSSING
CONTROL 3 08/05/95 00:06:13 14 FCOLL 13570 MANUAL PROCESSOR REMOVAL
CONTROL 15 08/05/95 00:06:13 12 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL
CONTROL 15 08/05/95 00:06:12 11 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL
CONTROL 15 08/05/95 00:06:12 10 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL
CONTROL 3 08/05/95 00:06:11 13 FCOLL 13570 MANUAL PROCESSOR REMOVAL
CONTROL 15 08/05/95 00:06:11 9 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL
CONTROL 15 08/05/95 00:06:10 8 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL
ADMIN 0 08/05/95 00:06:10 7 FCOLL 13570 EXTERNAL MEMORY REMOVAL
CONTROL 3 08/05/95 00:06:09 12 FCOLL 13570 MANUAL PROCESSOR REMOVAL
CONTROL 15 08/05/95 00:06:09 6 FCOLL 13570 MANUAL EXTERNAL MEMORY REMOVAL

FIG. 14

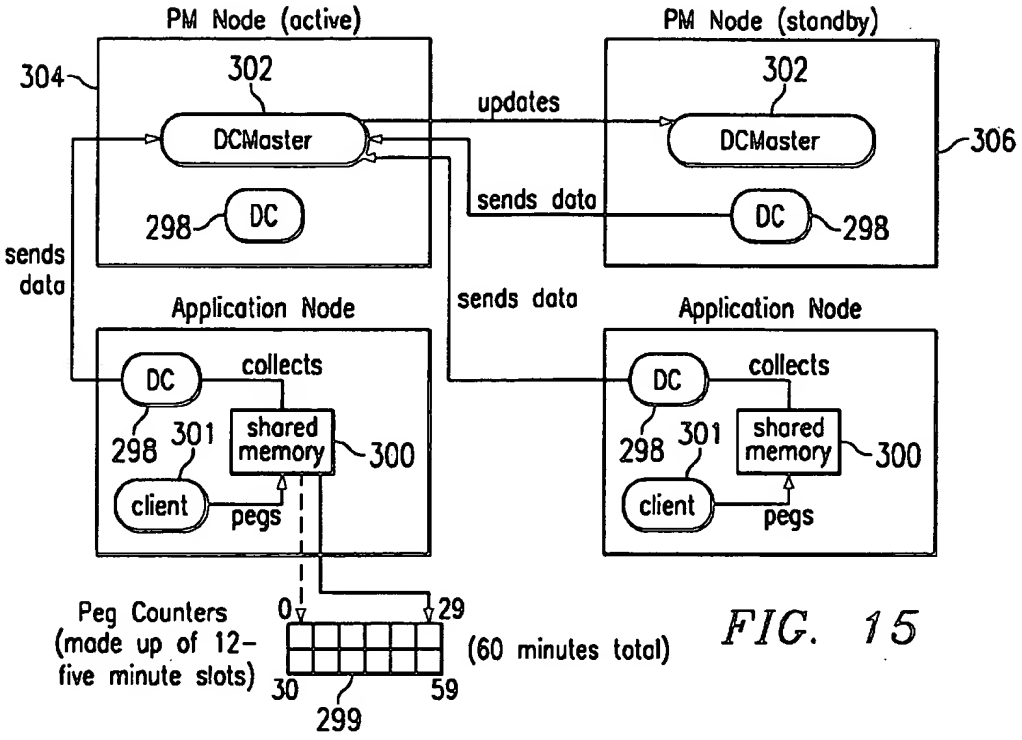

PM Node (active)
PM Node (standby)
304
302
302
updates
DCMaster
DCMaster
306
298
DC
sends data
DC
298
sends data
sends data
Application Node
Application Node
DC
collects
DC
collects
298
301
shared memory
300
298
301
shared memory
300
client
pegs
client
pegs
Peg Counters (made up of 12-five minute slots)
0
29
(60 minutes total)
30
59
299

FIG. 15

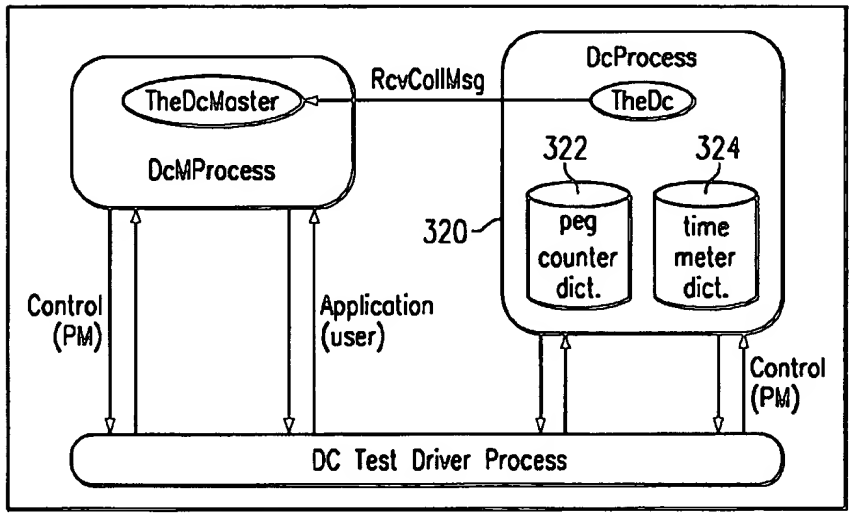

TheDcMaster
RcvCollMsg
DcProcess
TheDc
DcMProcess
322
324
320
peg counter dict.
time meter dict.
Control (PM)
Application (user)
Control (PM)
DC Test Driver Process

FIG. 16

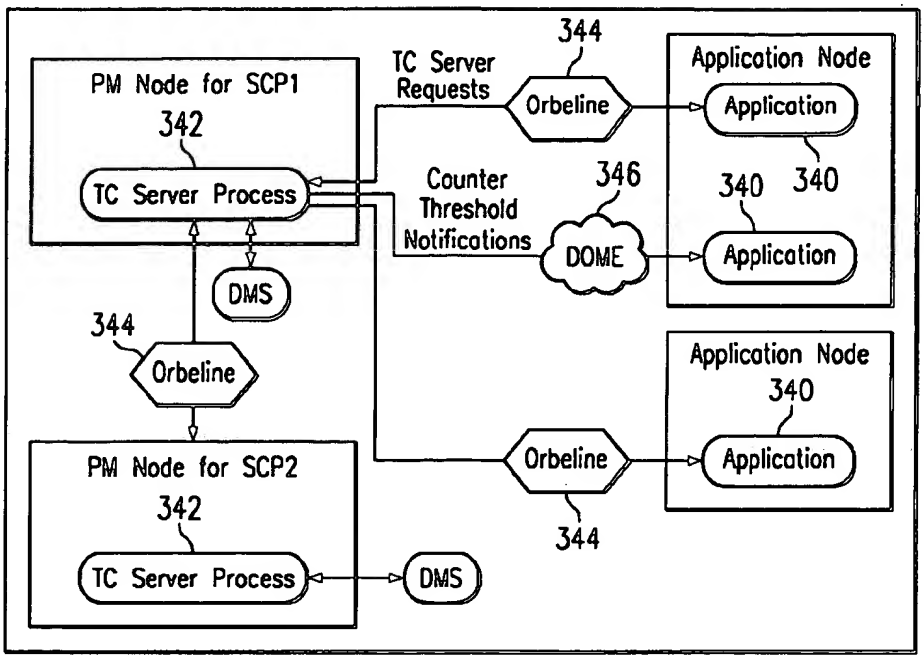

PM Node for SCP1
342
TC Server Process
DMS
344
Orbeline
PM Node for SCP2
342
TC Server Process
DMS
TC Server Requests
344
Orbeline
Application Node
Application
340
340
Application
Counter Threshold Notifications
346
DOME
Application Node
340
Application
Orbeline
344

FIG. 17

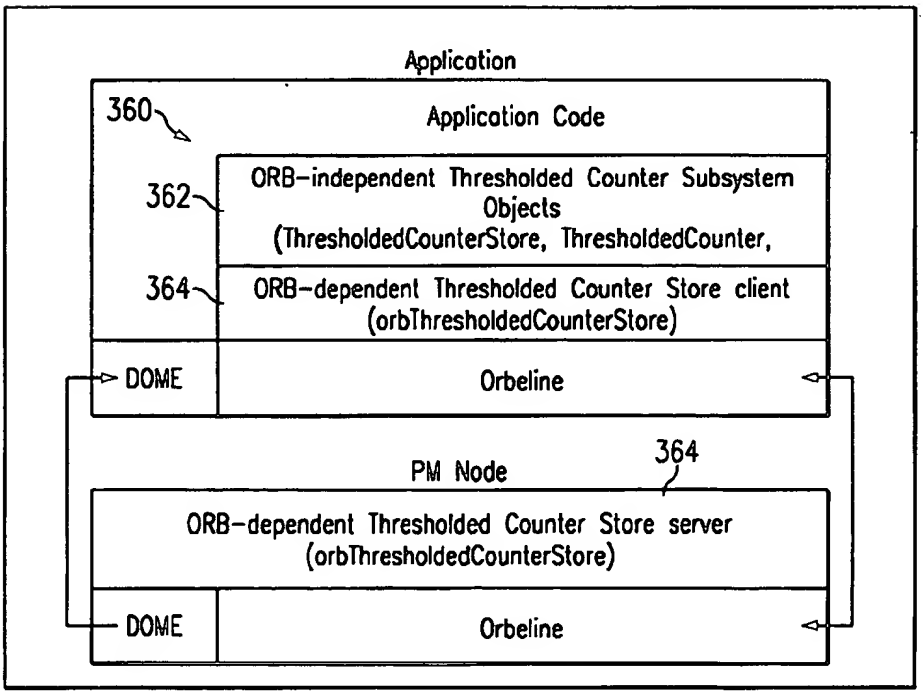

Application
360
Application Code
362
ORB-independent Thresholded Counter Subsystem Objects (ThresholdedCounterStore, ThresholdedCounter,
364
ORB-dependent Thresholded Counter Store client (orbThresholdedCounterStore)
DOME
Orbeline
PM Node
364
ORB-dependent Thresholded Counter Store server (orbThresholdedCounterStore)
DOME
Orbeline

FIG. 18

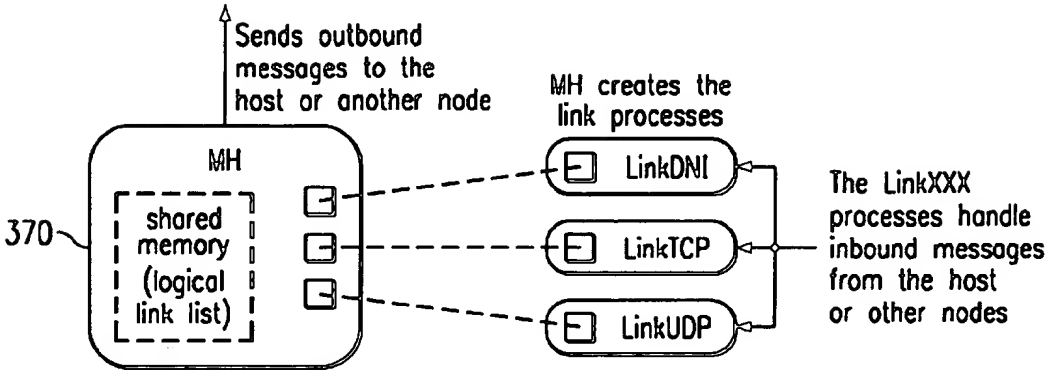

Sends outbound messages to the host or another node
MH creates the link processes
MH
shared memory (logical link list)
370
LinkDNI
LinkTCP
LinkUDP
The LinkXXX processes handle inbound messages from the host or other nodes

FIG. 19

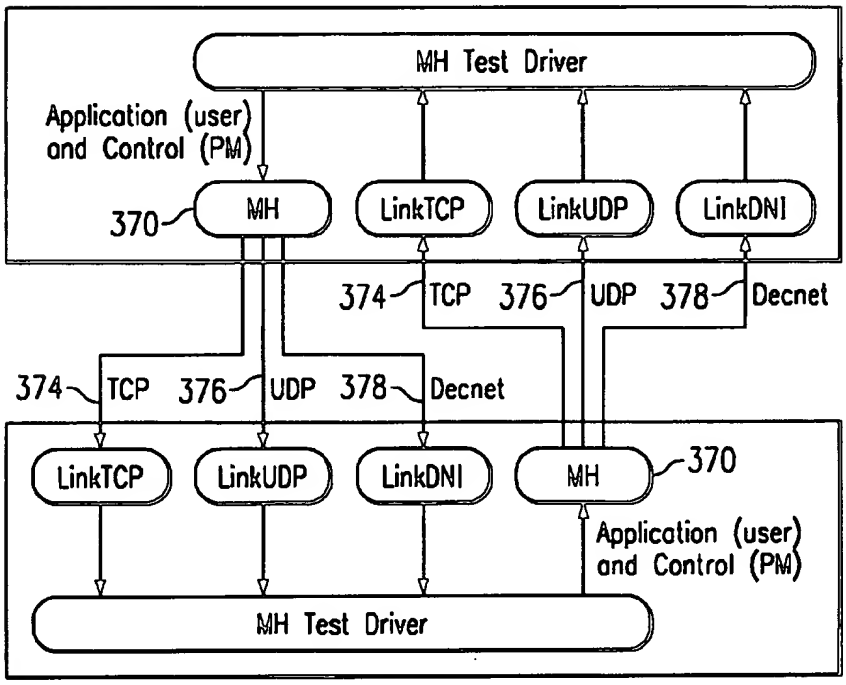

MH Test Driver
Application (user) and Control (PM)
370
MH
LinkTCP
LinkUDP
LinkDNI
374
TCP
376
UDP
378
Decnet
374
TCP
376
UDP
378
Decnet
LinkTCP
LinkUDP
LinkDNI
MH
370
Application (user) and Control (PM)
MH Test Driver

FIG. 20

FIG. 21

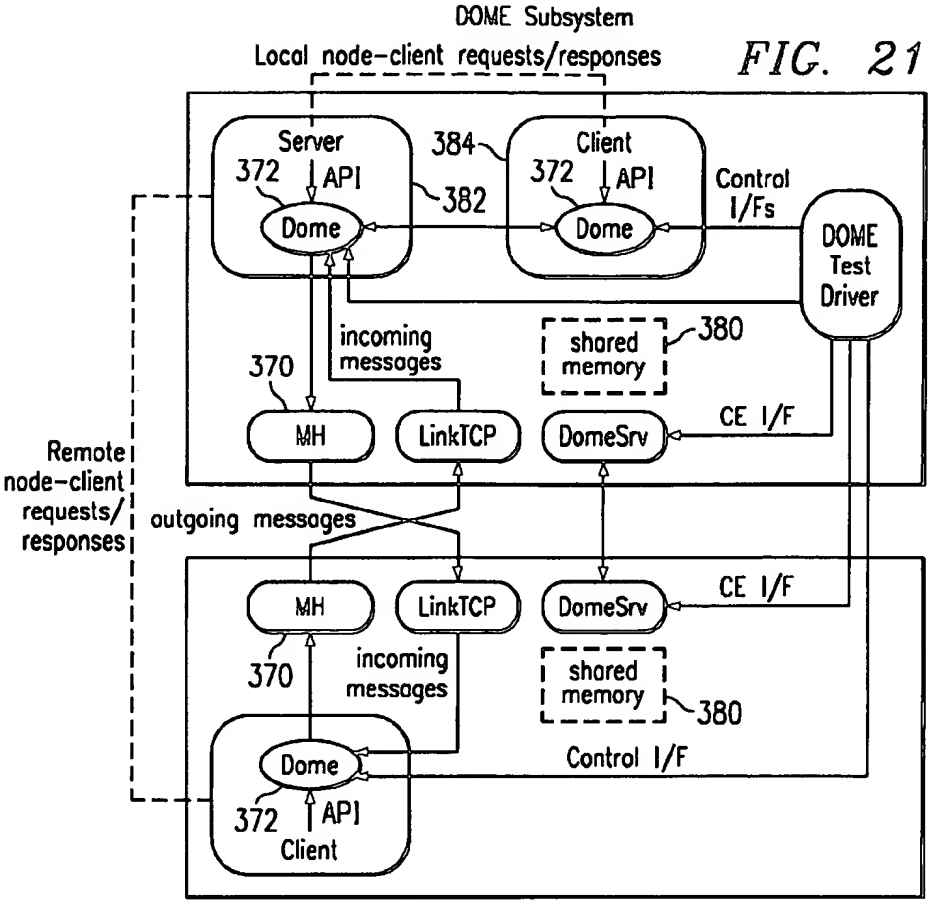
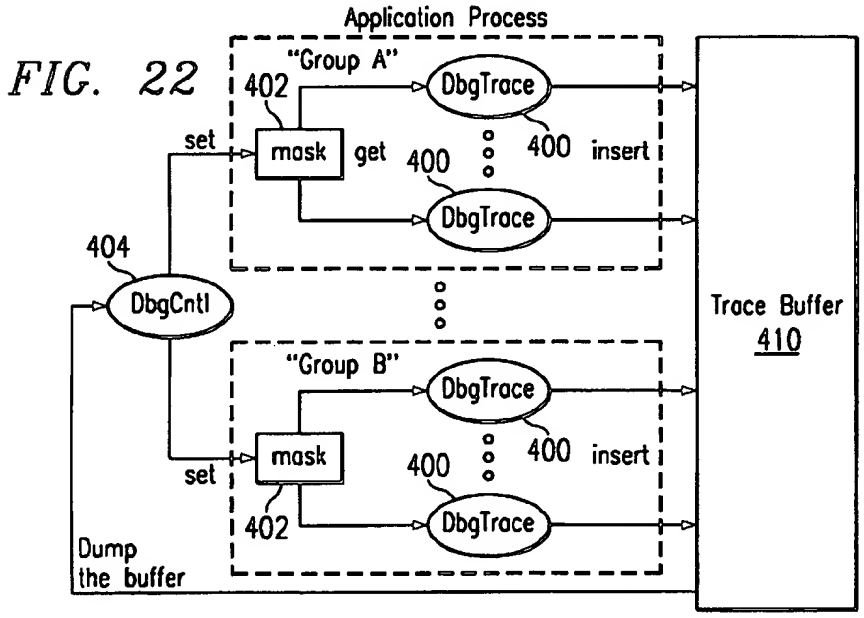

DOME Subsystem
Local node-client requests/responses
Server
372
API
384
382
Client
372
API
Control I/Fs
Dome
Dome
DOME Test Driver
incoming messages
shared memory
380
370
MH
LinkTCP
DomeSrv
CE I/F
Remote node-client requests/ responses
outgoing messages
MH
LinkTCP
DomeSrv
CE I/F
370
incoming messages
shared memory
380
Dome
Control I/F
372
API
Client

FIG. 22


Application Process
"Group A"
402
DbgTrace
set
mask
get
400
400
insert
DbgTrace
404
DbgCntl
Trace Buffer
410
"Group B"
DbgTrace
400
insert
mask
set
400
402
DbgTrace
Dump the buffer

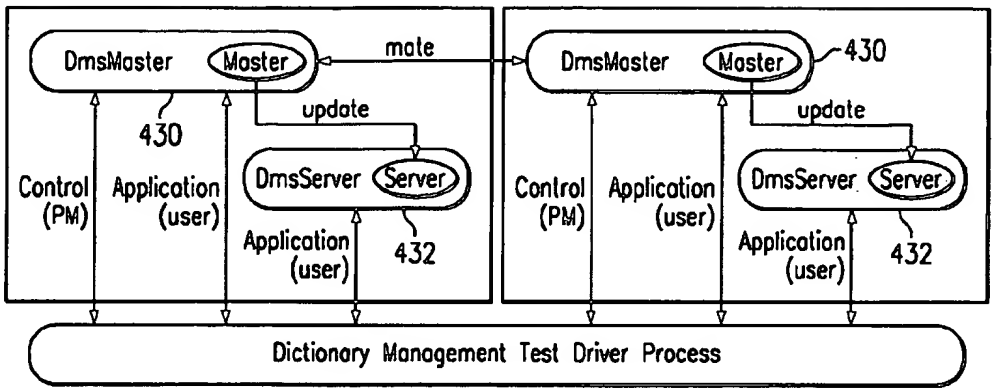


FIG. 23

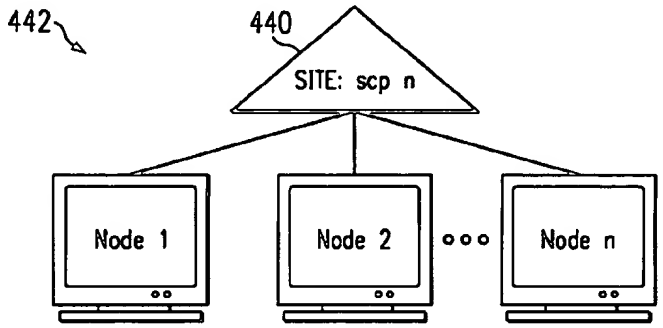


FIG. 24

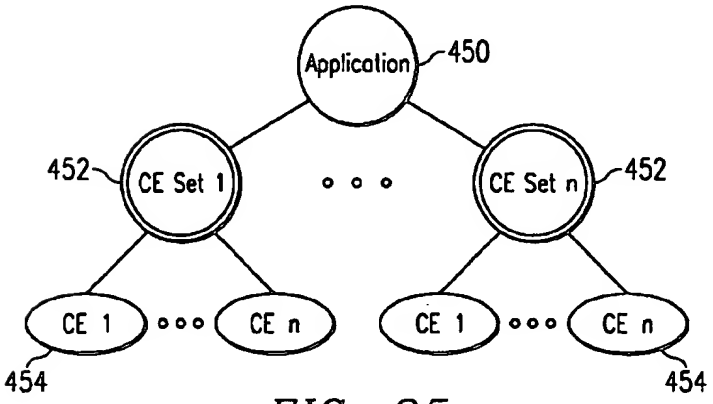


FIG. 25

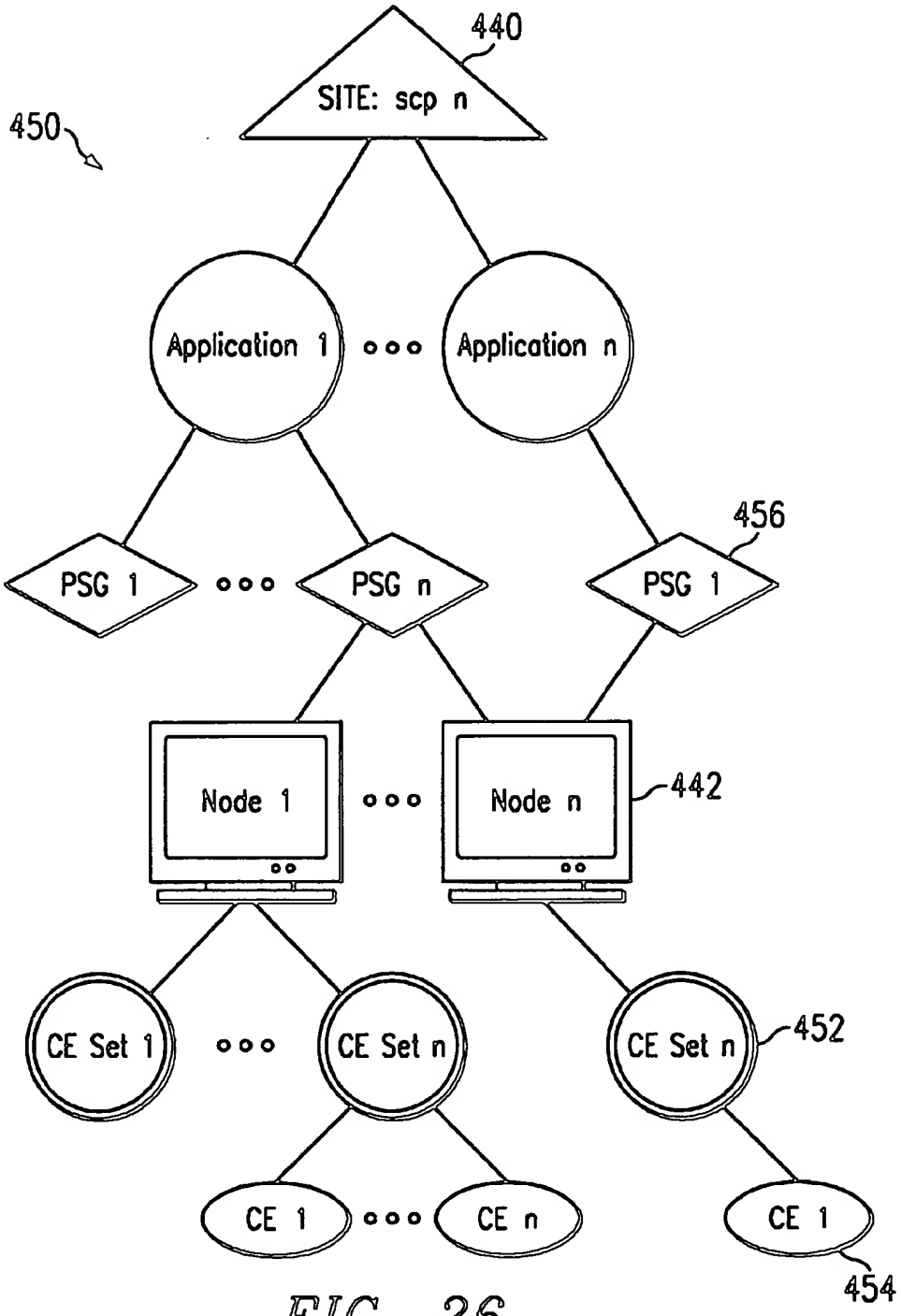

440
SITE: scp n
450
Application 1
Application n
456
PSG 1
PSG n
PSG 1
Node 1
Node n
442
CE Set 1
CE Set n
CE Set n
452
CE 1
CE n
CE 1
454

FIG. 26

METHOD AND PLATFORM FOR INTERFACING BETWEEN APPLICATION PROGRAMS PERFORMING TELECOMMUNICATIONS FUNCTIONS AND AN OPERATING SYSTEM

RELATED APPLICATION

This patent application claims benefit from provisional patent application No. 60/069,576, filed on Dec. 12, 1997, and entitled *Telecom Platform System and Method.*

TECHNICAL FIELD OF THE INVENTION

This invention is related in general to the field of telecommunications. More particularly, the invention is related to a telecom platform system and method.

SUMMARY OF THE INVENTION

In one aspect of the present invention, a telecom platform forming an interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network. The telecom platform includes network management processes operable to provide inter-node configuration, monitoring and management functionality, node management processes operable to provide node initialization, configuration, monitoring, and management functionality, event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events, common processes operable to provide a library of a plurality of programming tools for the development of the application programs, communications processes operable to provide message handling functionality, and distributed object processes operable to provide a distributed database repository for object-based communications.

In another aspect of the present invention, a method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network is provided. The method includes supplying network management processes operable to provide inter-node configuration, monitoring and management functionality, supplying node management processes operable to provide node initialization, configuration, monitoring, and management functionality, supplying event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events, supplying common processes operable to provide a library of a plurality of programming tools for the development of the application programs, supplying communications processes operable to provide message handling functionality, and supplying distributed object processes operable to provide a distributed database repository for object-based communications.

In yet another aspect of the present invention, a method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network is provided. The method includes providing a network platform manager operable to remove nodes from service, restore nodes to service, remove applications from service, and restore applications to service, providing a network system integrity manager operable to monitor the nodes and to enable failed nodes to recover, providing a configuration manager operable to interface with a host coupled to the telecom platform, providing a node platform manager operable to provide management functions for a node, providing a service manager operable to start and stop processes at the direction of the node platform manager, and providing a node system integrity manager operable to monitor inter-node links.

BRIEF DESCRIPTION OF THE DRAWINGS

For a better understanding of the present invention, reference may be made to the accompanying drawings, in which:

FIG. **1** is a simplified block diagram of the telecom platform architecture layers according to an embodiment of the present invention;

FIG. **2** is a simplified block diagram of the telecom platform conceptual components according to an embodiment of the present invention;

FIG. **3** is a block diagram of telecom platform's conceptual components and relationships therebetween according to an embodiment of the present invention;

FIG. **4** is a simplified block diagram of the logical partitioning of the telecom platform according to an embodiment of the present invention;

FIG. **5** is a simplified block diagram of the telecom platform services and their dependencies according to an embodiment of the present invention;

FIG. **6** is a simplified block diagram of the physical partitioning of the telecom platform according to an embodiment of the present invention;

FIG. **7A** is a block diagram of NetPM's testing flow according to an embodiment of the present invention;

FIG. **7B** is a block diagram of NetPM's time synchronization flow according to an embodiment of the present invention;

FIG. **7C** is a block diagram showing fault detection and interaction between network management services and node management services according to an embodiment of the present invention;

FIG. **7D** is a block diagram showing interaction between core services according to an embodiment of the present invention;

FIG. **8** is a state transition diagram of telecom platform nodes according to an embodiment of the present invention;

FIG. **9A** is a simplified block diagram of node start up process according to an embodiment of the present invention;

FIG. **9B** is a message flow diagram of node initialization process according to an embodiment of the present invention;

FIG. **9C** is a message flow diagram of node initialization process according to an embodiment of the present invention;

FIG. **9D** is a message flow diagram of node initialization process according to an embodiment of the present invention;

FIG. **10** is a message flow diagram of service management interface protocol according to an embodiment of the present invention;

FIG. 11 is a simplified block diagram showing Event Manager uses according to an embodiment of the present invention;

FIG. 12 is a simplified information and problem report (IPR) flow diagram according to an embodiment of the present invention;

FIG. 13 is a simplified IPR processing flow diagram according to an embodiment of the present invention;

FIG. 14 is an exemplary IPR view graphical user interface according to an embodiment of the present invention;

FIG. 15 is a simplified block diagram showing data collection according to an embodiment of the present invention;

FIG. 16 is a simplified block diagram of the data collection subsystem according to an embodiment of the present invention;

FIG. 17 is a simplified block diagram of the threshold counter data communication paths according to an embodiment of the present invention;

FIG. 18 is a simplified block diagram of the threshold counter subsystem according to an embodiment of the present invention;

FIG. 19 is a simplified block diagram of the message handling subsystem according to an embodiment of the present invention;

FIG. 20 is a simplified block diagram of message handling testing according to an embodiment of the present invention;

FIG. 21 is a simplified block diagram of the distributed object messaging environment according to an embodiment of the present invention;

FIG. 22 is a simplified block diagram of the internal debugging and tracing object relations according to an embodiment of the present invention;

FIG. 23 is a simplified block diagram of the dictionary management system according to an embodiment of the present invention;

FIG. 24 is a simplified block diagram of the hardware representation of the telecom platform according to an embodiment of the present invention;

FIG. 25 is a simplified block diagram of the software representation of the telecom platform according to an embodiment of the present invention; and

FIG. 26 is a simplified block diagram showing dynamic mapping of software onto hardware representation of the telecom platform according to an embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Architecture Overview

Telecom platform (TP) **10** of the present invention is a software system designed to support the development and execution of distributed, scalable, fault resilient telecommunications applications **12**. Telecom platform **10** provides a unique set of tools developed for a computing environment such as UNIX. These tools include not only the set of interfaces, libraries, and executables provided by the telecom platform development and runtime packages, but also a set of conceptual components necessary to design and manage distributed, scalable, fault resilient applications.

As shown in FIG. 1, telecom platform **10** is comprised of three distinct software layers **14–16**. Layer #1 is a telecom platform application programming interface (API) layer **14**; layer #2 is a telecom platform services layer **15**; and layer #3 is a systems interface layer **16**. Telecom platform API layer **14** provides the communication methods for accessing telecom platform services layer **15**, which is comprised of telecommunications middleware services. Telecom platform services layer **15** is the software layer that provides the most commonly needed middleware services for a UNIX-based telecommunications system, for example. System interface layer **16** is comprised of operating system (OS) API and the network links. System interface layer **16** defines the functions of process and thread management, memory management, timers, file system, communication, interface to hardware devices, and other system components. Telecom platform **10** allows higher level client applications **12** to be decoupled from the operating system and network. By using telecom platform **10**, developers may write applications without having to master the intricacies of the underlying services, such as the operating system and the network, that perform the work on behalf of the application.

FIG. 2 is a block diagram of the conceptual components associated with telecom platform **10**. The smallest conceptual component is a configurable element (CE) **30**. A configurable element **30** is defined by telecom platform **10** as one or more copies of a UNIX executable program that is administered by telecom platform **10**. For example, a configurable element may be a link process, database, graphical user interface, timing process, query process, error handlers, etc. Configurable elements **30** are the fundamental building blocks of application programs. The most basic services that telecom platform **10** provides to application developers are those service to create, configure, and monitor configurable elements **30**. Configurable elements **30** can be configured to be started at specific points during node initialization. The Unix executable configurable elements represent can be run multiple times for scalability or redundancy. Thresholds of the number of instances of configurable elements required to provide adequate services can be configured as well as whether or not the instances should be restarted automatically by the telecom platform **10** in the event of a process failure.

Configurable attributes of a configurable element includes RunLevel, which is the level a configurable element starts at. The RunLevels include PRE_MIN, OS_MIN, IN_SVC, and POST_IN_SVC. PRE_MIN run level specifies that the configurable element will be created automatically by a service management subsystem at boot time. PRE_MIN configurable elements are not monitored by the platform manager subsystem. OS_MIN specifies that the configurable element will be created when the node is transitioning to OS_MIN. IN_SVC specifies that the configurable element will be created when the node is transitioning to IN_SVC. POST_IN_SVC specifies that the configurable element will be created when the node transitions to the IN_SVC state. Another configurable attribute is NumberOfInstances, which specifies how many copies of the executable is to be run. InServiceThreshold is a configurable attribute that specifies how many out of NumberOfInstances is required to be up and running to make the configurable element's state be ENABLED. If the number of instances drop below this threshold, the entire configurable element or all the instances of the configurable element are removed. Another attribute of the configurable element is the HeartbeatSchedule which specifies the schedule for heartbeat messages to be sent to a configurable element. Each configurable element also has an AuditSchedule, which specifies the schedule for audit messages to be sent to the configurable element.

A configurable element set (CESet) **26** is defined by telecom platform **10** as a group of configurable elements

designed to be deployed together on one or more nodes **24**. A configurable element set is a distributable component. Telecom platform **10** may not manage configurable element sets **26** directly, but does support their creation and deployment. Configurable element sets **26** can be viewed as being the distributable and/or replicable components of an application **28**.

An application **28** is defined as a group of configurable element sets **26** that fully define all of the configurable elements **30** of a distributed program. Telecom platform **10** provides software to manage applications **28** within a site **20**. Defining the configuration of applications in terms of their distributable components allows the software for a distributed application to be defined independently of the hardware on which it will be run. An application's configurable element sets will at some point in time be deployed to the nodes **24** of a site **20**. When that occurs the scale and fault resilience of the application **28** will be determined based on the number of nodes used to support each configurable element set.

A node **24** is defined as an instance of a supported operating system on which telecom platform **10** runs. Telecom platform **10** provides software that manages processes on nodes **24**. Nodes **24** may be fault tolerant or non-fault tolerant, single or multi-processor. Telecom platform **10** uses the services of the operating system and is generally unaware of the hardware it is running on. Telecom platform requires very little configuration information for a node **24**. Nodes are configured into the system by providing their name and unique device identifiers.

Nodes **24** have operating states, supported by telecom platform, that describe the ordering of configurable elements started within them. The operating states includes HALTED, PRE_MIN, OS_*MIN, IN*_SVC, and POST_IN_SVC. The HALTED node state indicates that the operating system of the node has been shut down. The PRE_MIN state is used to start configurable elements that need to be started before configurable elements in the OS_MIN states are started. Telecom platform starts all configurable elements that are configured to run at PRE_MIN for that node first, then immediately begins running configurable elements that are configured to run in the OS_MIN state. Configurable elements that are configured to run at PRE_MIN do not directly effect the state of the node. The OS_MI node state coordinates all configurable elements configured for the OS_MIN run level will be started to bring the node to the OS_MIN state. All configurable elements configured for the OS_MIN node state achieve their configurable run-level transition state before the node is said to have transitioned to OS_MIN. Once the OS_MIN node state has been achieved, if any configurable element changes its state to be below its run-level transition state, the telecom platform will downgrade the node to the HALTED node state. A shut down node may recover automatically. The IN_SRV node state coordinates configurable elements configured for the IN_SRV run-level. All configurable elements configured for the IN_SRV node state achieve their configurable run-level transition state before the node is to have transitioned to IN_SRV. Once the IN_SRV node state has been achieved, if any configurable element changes its state to be below its run-level transition state, the telecom platform will downgrade the node to the OS_MIN node state. Automatic recovery of a node may occur if the node downgrade was not originated manually. The POST_IN_SRV node state is used to configure configurable elements that are to be started immediately after a node has transitioned to IN_SRV. Once a node has achieved IN_SRV, the telecom platform creates each POST_IN_SRV configurable element. State changes for POST_IN_SRV configurable elements do not affect node state, and may be started and stopped repeatedly. The process of stopping a POST_IN_SRV configurable element does not cause the node to downgrade to a lower node state.

A site **20** is defined by the telecom platform to be a group of nodes that distributed applications can be deployed across. Telecom platform provides a telecom platform application known as the platform manager that manages nodes **24** within a site **20**. A site may be made up of at least one node. In multi-node sites, the platform manager application may run as an active/standby distributed application in two of the nodes. In single node sites, the platform manager application runs in the single node along with user defined applications, but runs without the fault handling capabilities provided by a standby node. Administration of a site is provided through the platform manager.

A processor service group (PSG) **22** is defined as a group of nodes that a specific configurable element set **26** is deployed to for redundancy. Telecom platform **10** provides software applications to manager processor service groups within an application. Processor service groups support redundancy by allowing the telecom platform user to identify the number of nodes a configurable element set is required to run on to provide an adequate level of service. As the state of the nodes or the configurable element sets running on them change, telecom platform **10** verifies that the appropriate level of service is maintained or it will change the application status as configured.

FIG. **3** is a diagram illustrating a system **40** design employing the conceptual components of telecom platform **10** which are mapped onto hardware components.

In terms of hardware configuration, a node is a computer processor within a network (such as ethernet) that can act either as a client or a server. Each node has a single instance of the operating system running on it. The processors within a node cannot run independently from one another because of their dependence on the operating system. Each node at a site can be classified as a platform manager or an application node. A site can consist of one node or a grouping of nodes that are connected to a host. The platform manager node has a redundant mate. The platform manager node and its mate may operate in an active/standby mode or a load-sharing mode.

System **40** has eight nodes, which includes two platform manager nodes (active **42** and standby **43**) and six application nodes **44–49**. An application **50** for handling telephone calls based on the time the call is placed, or time dependent routing, is deployed across the nodes. Configurable element sets **52** and **54** of application **50** are the distributed components which supply the time dependent routing functionality. Each configurable element set **52** and **54** contain the software processes of the UNIX executable programs or configurable elements for a specific time zone. As shown, application **50** does not have to reside on a single application node **44–49**. It may be desirable to map configurable element sets onto different nodes. This makes it possible to scale the application by increasing the number of nodes to which the configurable element sets are configured.

The telecom platform internal architecture is described from both the logical and physical partitioning perspectives. The logical partitioning decomposes the telecom platform into distinct functional areas as shown in FIG. **4**. Each functional area contains a cohesive group of classes, which together provide one particular system function. The physical partitioning describes the concrete software and hardware decomposition of the system's context. The services

provided by telecom platform **10** may be partitioned into two groups: application services **60** and core services **62**. Application services may include services that perform information and problem report (IPR)/alarm **64**, statistics **65**, dictionary **66**, graphical user interface (GUI) **67**, and host maintenance simulator (HMS). IPR/alarm services **64** provide a standard mechanism to inform the system user of error conditions and other pertinent system information. Statistics services **65** provides the methods to access system-wide measurement data and to generate reports based on the collected data. Dictionary services **66** provide classes that are designed to support data storage (persistent, shared or private) and access to the data. Graphical user interface services **67** provide primitive abstractions for building GUI applications, and access to system utilities and to the system itself, e.g., xterm window and operating system utility programs. Host maintenance simulator services **75** provide a method of interfacing with the telecom platform when there is only one node within the system or when there is not a host to which to connect. It is through the host that control and operation of the platform is made possible.

Core services **62** may include services that perform network management **68**, node management **69**, distributed object **70**, communications **72**, common functions **73**, and event handling **74**. Network management services **68** directs network activities, e.g., configuration of nodes and network-level fault processing. Node management services **69** directs node-level processes, e.g., node status reporting and link management. Distributed object services **70** provide a distributed database repository for object-based communication in a multi-processing environment. Communications services **72** provide the mechanism for handling messages across interprocessing links external to the platform. Common services **73** provide a library of programming tools to aid in the rapid development of processes designed to run on or within the telecom platform. Event services **74** provide the capability to initiate, terminate, and/or distribute specific actions significant to a task.

As a minimum, telecom platform provides all of the core services. High level applications use these services to accomplish the lower level functions.

FIG. **5** further shows the telecom platform services and their dependencies. The developer accesses all of the core and application services through telecom platform application program interfaces **14**. The developer may also access the operation system, network, and third party software/hardware if the need arises. Interprocess object-based communication is handled by communication services **72**. Most of the core and application services dependent on communication services **72** and common services **73** to perform their respective functions. Graphical user interface services **67** may only be dependent on communication services **72**. The arrows in FIG. **5** indicate the dependency relationships between the services.

FIG. **6** is a diagram of the physical partitioning of telecom platform **10** which includes an application layer **80** and a core layer **82**. Core layer **82** containing core services **62** exists for every instance of a telecom platform. Core layer **82** contains telecom platform API **14**, interprocess communication mechanisms, event mechanisms, and platform management. Telecom platform applications layer **80** has both vertical and horizontal partitions. Vertically, each telecom platform application process is classified as either a part of a main set of applications **84** or not. Non-main set processes are dependent on the main set processes. Horizontally, telecom platform applications **80** are categorized as required or optional. Optional applications may include an IPR/alarm package **86**, a data collection package **87**, a dictionary management system package **88**, and a host maintenance simulation package **89**.

The following is a more detailed description of Telecom platform services.

Network Management Services

Network Management services **68** provides a common administrative view of the network element. It is responsible for implementing high level operations on the network element nodes such as removing server nodes from service, restoring server nodes to service, removing applications from service, restoring applications from service, enabling or disabling applications, maintaining status of distributed applications, maintaining server node state and status, and reporting application status changes. Network management services **68** includes a network platform manager (NetPM), network system integrity subsystem (NetSI), and configuration manager (ConfigMgr). FIG. **7A** is a block diagram showing an active platform manager node **100** with a corresponding or mated standby platform manager node **102**. Each platform manager node includes a network platform manager **104**, a network system integrity subsystem **106**, and a configuration manager **108**. A platform manager network test driver **110** provides network level testing.

Network Platform Manager (NetPMMain)

The class name for the network platform manager is NetPM. NetPM is responsible for providing management functionality of the platform resources. The platform is a distributed system consisting of multiple nodes or servers which provide processing power for specific services, such as calling card or credit card validation. The service provided by a server is determined by the configurable elements residing on the node. NetPM manages all the configuration data associated with the platform. Configuration data includes information about the hardware, such as the TCP/IP address of a server, status information, such as server and query status, software configuration information, such as application type, node name, and information relating to the individual configurable elements.

NetPM maintains the following configuration information. This information is collected by NetPM during its initialization.

- Configurable element descriptor information—This provides configuration information for each Configurable element of the platform. NetPM retrieves these from a disk file containing the information on configurable elements of different types.
- Application information—This provides configuration information about each application (service), which can be used in calculating an application's status. NetPM retrieves this information from a disk file containing the information for all the applications in the platform.
- Processor service group information—This provides configuration information about Processor service groups, which can be used in calculating the Processor service group status (Processor service group designates group of processors serving the same application, i.e., CCD, CCL). NetPM retrieves these from a disk file containing the information for all Processor service groups in the platform.
- Server information—This provides specific information about all servers in the platform. NetPM requests and retrieves this information from the ConfigMgr. ConfigMgr provides NetPM with the server information on platform manager nodes first. Afterwards if ConfigMgr determines that the current server is the active platform manager, it provides the local NetPM with the infor-

mation on the remaining servers in the platform. Otherwise (standby platform manager), NetPM will retrieve those information from its mate, and not from the ConfigMgr.

If an error is detected while collecting these information, NetPM issues appropriate IPRs and exits.

NetPM uses a NetMAP object to manage all the configuration data. NetPM also uses a persistent dictionary to retain server status, query status, and scheduled actions information across platform manager resets. A Disk File Dictionary object is used to manager this dictionary. NetPM is responsible for maintaining the integrity of the configuration data between the two platform manager servers. NetPM uses a persistent dictionary, database equalization, and auditing to maintain the integrity of the data.

Application status is determined based on the processor service group status. The following criteria is used in determination of the processor service group status:

PSG_DISABLED—At least a set number of servers in the processor service group are in disabled state.

PSG_INACTIVE—At least one server in each processor service group is in standby state, and none is in active state.

PSG_ACTIVE_MINIMAL—Only certain number of servers in the processor service group are in active state.

PSG_ACTIVE—A set number of servers in the processor service group are in active state (Note: This number will be greater than the number of servers that need to be active for PSG_ACTIVE_MINIMAL.)

and the application status may be derived using the following criteria:

AP_DISABLED—At least a set number of processor service groups for the given application have status of PSG_DISABLED.

AP_INACTIVE—At least one processor service group for the given application has status of PSG_INACTIVE, and no processor service group has status of PSG_ACTIVE.

AP_ACTIVE_MINIMAL—A set number of processor service groups for the given application have status of PSG_ACTIVE_MINIMAL or higher (PSG_ACTIVE).

AP_ACTIVE_PARTIAL—A set number of processor service groups for the given application have status of PSG_ACTIVE_MINIMAL or higher (PSG_ACTIVE) (NOTE: The number of processor service groups required for AP_ACTIVE_PARTIAL state is greater than required number of processor service groups for AP_ACTIVE_MINIMAL).

AP_ACTIVE—A set number of processor service groups for the given application have status of PSG_ACTIVE (NOTE: The number of processor service groups required for AP_ACTIVE stat is greater than required number of processor service groups for AP_ACTIVE_PARTIAL).

NetPM keeps track of the status changes on each server node, and as it gets them it determines the status of the processor service group and in case of a change, determines the new application status for the node, and informs ConfigMgr of these changes.

NetPM provides solicited and autonomous updates on application status. For autonomous updates, the application process first registers a function with NetPM to receive updates for a particular application type (CCD or CCL). Whenever NetPM receives a change of server or query status from NodePM, the application status is calculated and the registered function is called with the old and new application statuses. Application status can also be solicited, during which NetPM will return the latest calculated value of application status saved in its NetMAP to the requesting process.

NetPM provides, partially through the use of two alias objects, two sets of routing options to other processes wishing to communicate with NetPM. NetPM provides a local, and a global active-standby option. In the local option, all NetPM client requests are sent to the NetPM server object in the same node as the client object. In the global active-standby option, all NetPM client requests are sent to the globally (i.e. possibly inter-nodal) available active NetPM server object.

NetPM provides a set of reader, and writer, functions for a lot of the Server configuration data. These include reader/writers for the schedule action data, the platform manager active status data, the server status data, etc. NetPM provides no direct read/write operations for the configurable element description data.

NetPM also provides a function to initialize the majority of the Server configuration data. This function expects a ServerInfoMsg object as input.

NetPM provides a set of functions which cause a specific configuration action (such as graceful halt, immediate halt, graceful downgrade, and restore), to occur on a specific Server.

NetPM provides a function where the server status can be changed on a specific server.

NetPM provides a function to enable, and a function to disable the query processing on a specific server.

NetPM provides several functions which "report" server status, and query status changes. These routines save the new status information in NetMAP, notify the ConfigMgr software of the change, and broadcast the change to all the NodePM software in the platform.

NetPM is also responsible for time synchronization within the server network. Time synchronization consists of three major parts, as shown in FIG. 7B. The first part is for active platform manager **100** to equalize its local time with the time of the host. This includes converting the host's **(110)** time into a usable form and informing the NodePMs **112** on platform manager nodes **100** and **102** to perform an adjtime( ) function to adjust their clocks in line with host **110**. NetPM **104** also informs the host ticker class of the new host time when it receives the time message. An xntp process **120** then synchronizes the application nodes' **(121)** time with the time of the platform manager nodes **100** and **102**. Each of the platform manager nodes **100** and **102** are configured as xntp master sources of time. The xntp daemon slaves **122** on application nodes **121** choose one of the master xntp daemons **120** on platform manager nodes **100** and **102** to keep in synch with. Finally, whenever an unsolicited Set Time message is received from host **110**, the network's time is the same as the received time.

Lastly, NetPM **104** provides a function which provides a newly booted node with pertinent server configuration data of all the servers in the platform. NetPM **104** is a configurable element. NetPM **104** provides the unencapsulated operations: Remove, Restore, and GetStatus which NodePM requires to control NetPM's execution. NetPMTimerHandler is called when the audit timer fires. It aborts the provide service loop and calls the NetPM function SettimeToVerify to start the audit.

NetPM **104** is an object with its own thread of control. After building up its NetMAP lists, NetPM **104** goes into an

infinite loop waiting for requests. NetPM **104** notifies ConfigMgr **108** whenever there is a change in the service or query status of a server. NetPM **104** also sends these status changes to all the NodePMs **112** in the platform. NetPM **104** notifies the specific NodePM **112** to enable, or disable, query processing. NetPM **104** provides service status synchronization functionality. NetPM **104** builds up the IPU information for the servers in the platform and passes this information to the specific NodePM **112** in the BootNotify member function. NetPM, in all the configuration requests for degradation of service (i.e. GraceDown, ImmedDown, GraceHalt, and ImmedHalt), notifies the specific NodePM **112** of the desired state of the server. NetPM **104** does several things when a server restore is requested. First, NetPM **104** obtains the current status of the server from the specific NodePM **112**. Second, if the returned status is out-of-service/minimum-software, NetPM **104** sends the specific NodePM **112** the relevant NodeSpecInfo. Third, NetPM **104** sends the relevant configurable element descriptor information to the specific NodePM **112**. Lastly, NetPM tells the specific NodePM to restore to service.

Network System Integrity (NetSIMain)

The Network System Integrity (NetSI) subsystem **106** provides monitoring and recovery operations for the network element. It is responsible for implementing network monitoring and recovery. Operations implemented by Network System Integrity include:

platform manager active/standby status monitoring

node failure report correlation

failed node recovery actions

The class name of Network System Integrity is NetSI. NetSI **106** manages network system integrity for the platform manager. NetSI **106** receives notifications of server downgrades and communication faults from the NodeSI on the faulted node. NetSI **106** determines what action should be taken based on the data given by NodeSI. If the node indicates a downgrade, NetSI will take the appropriate action to downgrade the node from the network level to the desired downgraded state. If the node indicates a communication fault, NetSI **106** will determine what node (if any) is at fault from data received previously and will take action to downgrade the faulted node if necessary. When NetSI determines that a downgrade is required for a node, NetSI calls the appropriate NetPM operation to perform the downgrade. If a change in active status is required, NetSI calls the appropriate NetPM operation to set the active status. After NetPM is called to perform the downgrade, NetSI notifies ConfigMgr that the status is changing for a particular node. This allows the host to be informed immediately that a node is being downgraded. NetSI then writes an entry to the network configuration report indicating the status change and reason for it. NetSI downgrades nodes to the legal service state based on the current state of the node.

NetSI contains a communication fault list. This list holds the reporting server node name and problem server node name of each communication fault report received. When a communication fault report is received, the list is searched for another report about the problem node. If not found, the fault information is added to the list. NetSI also contains a down status info list. When NodePM indicates that a node is out of service and the NetPM status does not indicate the node is halted, a down status info entry is created with the node name of the halted IPU. A timer is created and the down status info is added to the list. If NodePM later indicates a higher status for that node (before the timer expires), the down status info entry is cleared from the list and no further action is taken.

NetSI routinely audits the status conditions of both PMs. If invalid conditions are present, NetSI attempts to correct the situation by setting the active status to the correct state. Other processes can also request NetSI to audit the platform manager status conditions.

NetSI operates with a "send to both" load shared concept. If both platform manager nodes are operational, each NetSI process on each platform manager node will receive the NodeSI request. Each NetSI process will determine if it should handle the request based on the platform's active/standby state and faulted server. The active platform manager's NetSI process will usually take the required action while the standby platform manager discards the information. However, if the faulted node is the active platform manager, the standby platform manager(if valid) will set itself to active and take the request action to downgrade the other platform manager node.

Each time a NetSI operation is called, NetSI first determines if it is the active or standby platform manager. If active, NetSI will process the request for all conditions except when the target node is itself and the mate is in service. If in standby, NetSI will discard the request for all conditions expect when the target node is the mate.

During initialization NetSI requests the mate's node name and server descriptors of its own server and mate server from NodePM. Before requesting the information, NetSI polls for the status of NodePM, and will not request the node name and server descriptors until NodePM is read to provide them. NetSI will not be ready to provide service until this information is received properly.

NetSI uses the command line parameter DWN_RPT_FILE to get the name of the network configuration (downgrade) report file name. If this parameter is not specified, no report entry is made of the downgrades.

Referring to FIGS. **7C** and **7D**, process interaction between node management and network management is shown. Constant monitor (ConMon) **132** is an instance of an object running on an application node **136**. ConMon **132** detects a faulted process or a failed configurable element, it notifies a service management process program **134**. Service management process **134** determines if the configurable element failure causes the process to fall below its threshold level. If it does not, the service management process **134** restarts the configurable element. However, if the configurable element does fall below its threshold level then service management process **134** generates a configurable element status change message and forwards the notification to NodeSI **130**. NodeSI forwards the configurable element status change to NodePM **112**. NodePM **112** determines whether the configurable status change affects the run level of the node, which could cause a downgrade of the node. If the node is to be removed, NodePM **112** provides instructions to service management process **134** to remove all of the configurable elements necessary to achieve the downgraded state. NodePM **134** notified the NetPM **104** of the node status change. NetPM **104** performs a calculation to determine if the node status change affects the processor service group and application status. NetPM's calculation also determines if an auto-action, such as removing a node from in-service to min-set and restoring it again, should be performed on the node. If the node is to be removed, then the node status change is forwarded from NetPM to ConfigMgr **108**. ConfigMgr notifies host **140** of the state change for the node, processor service group, and application. These state changes can be displayed or printed in a report.

In particular, each NetSI determines if it should handle the downgrade request. If so, the target server's status is

retrieved. If the target server is not already halted, the server is downgraded to the appropriate status based on the IPU status. If the IPU status is out of service, NetSI calls NetPM's immediate halt operation to either auto halt or manually halt the target node. If the IPU status is Out of service minimal (OS-MIN), NetSI calls NetPM's immediate downgrade operation to downgrade the target node to OS-MIN. If the IPU status is in service disabled, NetSI calls NetPM's disable query operation to disable query status for the target node. In all cases, NetsSI updates the active status if the target node is the active platform manager. Also, if the target node is part of the local site, NetSI informs the host via ConfigMgr that a status change is occurring and initiates recovery of the processor service group (through ConfigMgr) if it determines that the processor service group of the target server should be recovered. NetSI then writes an entry to the network configuration report file indicating the status change is occurring due to the node reporting a fault.

NodeSI informs NetSI of communication faults that occur between two nodes. NetSI stores or takes action on the fault based on previous information receive (if any). Each NetSI determines the status of the reporting and problem nodes. If either server is halted, the communication fault report is discarded since the integrity of the data cannot be assured. If neither server is halted, the Communication Fault List is searched for another report on the problem node. If no report on the problem node is found, a Communication Fault List entry is added to the List with the server information. If another report of the problem node is found and another reporting server has reported it, the problem server is set up for downgrade processing. Once a decision is made about whether the server should be downgraded, NetSI determines if it should handle it (based on its active state and whether or not the target server is itself.) If it should handle the downgrade, NetSI calls NetPM's Immediate Halt operation to either Auto Halt or Manually Halt the problem node. If the server to be halted is the active PM, NetSI updates the active status accordingly before halting the node. Also, if the target is part of the local site, NetSI informs the Host via ConfigMgr that a status change is occurring and initiates recovery of the Processor service group (through ConfigMgr) if it determines that the Processor service group of the target server should be recovered. NetSI also writes an entry to the network configuration report file indicating the halt is occurring due to a communication fault.

Configuration Manager (ConfigMgr)

The Configuration management subsystem (class name: ConfigMgr) provides the control interface between the SCP Host and Server components. All operations that can be performed on the server network are defined in this interface. The Configuration Management subsystem implements the following features:

Control Message Interface between Host and Servers

State Machine for valid operations

Drives Network Management with requests.

Controls operation timing/timeouts.

ConfigMgr manages server configuration control for the platform manager. ConfigMgr receives Host messages transmitted on the CONFIGCTL, MAINT, APPLCTL and ROUTINGCTL logical links and processes each based on its message id and type. If the Host requires a response or report to be sent, ConfigMgr determines the necessary response and retrieves the necessary report information and sends it back to the Host. ConfigMgr handles the following messages:

APPL_STATUS_MSG
ASPEC_MSG
CONFIGURE_SERVER_MSG
PSG_INFO_MSG
PSG_STATUS_MSG
QUERY_PROCESSING_MSG
RESET_SERVER_MSG
ROUTING_INFO_MSG
SCHED_ACTION_CTL_MSG
SERVER_INFO_MSG
SERVER_STATUS_MSG
TEST_SERVER_MSG
TIME_MSG

ConfigMgr also provides operations to the platform manager for retrieving server and time information from the host. It also provides operations to notify the host of server status changes. In processing host command messages, there are times when ConfigMgr must wait for a response from the host or for a status change from a particular server. ConfigMgr uses a non-blocking philosophy in respect to these waits. Instead of stopping and waiting for the event to occur, ConfigMgr saves the desired response or status on a PendingQueue and continues normal processing of another Host message or providing service to a client. When the desired response or status occurs, the appropriate procedure is called to resume processing of the host commanded message. If the desired response does not arrive or desired status does not occur within the specified time limit, a fail procedure is called to clean up processing of the Host commanded message and issue IPRs as needed.

In addition to processing host command messages, ConfigMgr is required to notify the host when a status change occurs. When ConfigMgr is notified of a status change, it checks the status pending queues to determine if it is waiting for the status change to occur. If so, the pending queue success operation is performed. Otherwise, ConfigMgr sends server status messages to the host. In processing host response messages, ConfigMgr checks the host response pending queue (HostPendQueue) to determine if it is waiting for the response. If so, the pending queue success operation is performed. Otherwise, ConfigMgr discards the response message from the Host. When a platform manager node is booted to OS-MIN state, it audits its mate and determines the status of the mate. In the event that no mate platform manager node is present, the mate status is automatically set to halted. Similar audits are done on service server nodes (nodes other than PM) to determine their status.

ConfigMgr has a registration capability where a subsystem can register to provide routing information for a particular application. When the Host requests routing information about an application, ConfigMgr makes a request to the appropriate registered subsystem (if one exists) to provide the routing info.

Configure Server Messages (ConfigServerMsgs) require special processing due to the nature of the services that are performed (i.e. halts, downgrades, restores, and boots). Since host messages are sent to both platform manager servers, care must be taken to assure that only one platform manager node processes the request. This requires checking the server state of the platform manager node and its mate. There are different actions to be taken based on the server stats of the platform manager nodes and whether the ConfigServer request is for a platform manager node, its mate, or a service server. Two finite state machines (PMCfgSvrFSM and SvcCfgSvrFSM) manage all the different state driven actions.

PMCfgSvrFSM is the finite state machine that handles the restores, halts, resyncs, downgrades, and boots for a platform manager application server. This machine processes a request based on whether the request is for itself or its mate, its own status, its mate's status, and the event requested (halt, downgrade, restore, etc.) The platform manager server states checked are: Halted (Auto), Halted (Manual), XOS-MIN, AOS-MIN (Auto), MOS-MIN (Manual), and In-Svc. If In-Svc, the active/standby status is checked to determine if the server is active or standby. Valid events are Restore, Graceful Halt, Immediate Halt, Graceful Downgrade, Immediate Downgrade, Graceful Boot, Immediate Boot, and Host Resync.

The event is important for determining which platform manager node will process the request. If a restore is requested, normally the platform manager node which is being restored will process the restoration (i.e. a platform manager node will restore itself). Processing a restore request a platform manager server that is halted, the halted server's mate (if able) will send a Denial response back to the host. If any Halt, downgrade, or boot is requested for a platform manager node, the platform manager node's mate will process it, unless the mate is halted. When the mate is halted the platform manager node will process the halt, downgrade, or boot for itself. Processing a halt, downgrade, or boot may involve actually performing the requested action or sending a Denial response back to the host. If a halt, downgrade, or boot request is not denied, the host considers the action successful.

When a platform manager node has to process a boot for itself, the platform manager node calls the GraceHalt or ImmedHalt operations (based on Boot type) of NetPM to bring itself into a halted state. Processing is then complete for this node since it is being brought down to a halted state. (The host will initiate the reset and boot of the server.) A force flag is checked when a halt, downgrade, or boot is requested for the last In-Service platform manager node. If the force flag is not set, the request will be denied with a response of "DENIED-LAST AMP". If the force flag is set, the halt, downgrade, or boot will be performed on the last In-Service platform manager node.

If a Host Resync is requested for a platform manager node, the target platform manager server's mate will process the request unless the mate is halted. If the target platform manager server's mate is halted, the platform manager node for resync will process the request. Processing the request involves changing the server status from XOS-MIN to AOS-MIN or MOS-MIN or denying the request if the current status is not XOS-MIN.

SvcCfgSvrFSM is the finite state machine that handles the restores, halts, resyncs, downgrades, and boots for a Service application server. This machine processes a request based on the state of the platform manager node performing the action, the state of the service server being worked on, and the event requested (halt, downgrade, restore, etc.) The service states checked are Halted (auto), Halted (manual), XOS-MIN, AOS-MIN (auto), MOS-MIN (manual), and InSvc. Valid events are Restore, Graceful Halt, Immediate Halt, Graceful Downgrade, Immediate Downgrade, Graceful Boot, Immediate Boot, and Host Resync.

The active platform manager node (OS-MIN or In-Service) will process the configure server request for a Service server. A boot, halt, resync, or downgrade is allowed on a service server as long as one platform manager is at least OS-Min. A restore for a service server is only allowed when at least one platform manager is In-Service. If neither platform manager node is In-Service, the platform manager node that is active will send a DENY-AMP not In-Service response back to the host. If a halt, downgrade, or boot request is not denied, the host considers the action successful.

A force flag is checked when a halt, downgrade, or boot is requested for the last In-Service node of an application. If the force flag is not set, the request will be denied with a response of "DENIED-LAST SERVER IN Processor service group PROCESSING QUERIES". If the force flag is set, the halt, downgrade, or boot will be performed on the last In-Service node of the application.

An Under Configuration flag is checked whenever a configure event (except Immediate Halts) is processed. If the Under Configuration flag is set, the request will be denied with a response of "DENIED-SERVER UNDER CONFIGURATION". ConfigMgr sets and clears the Under Configuration flag during event processing. The other messages (i.e. ServerInfoMsg, ServerStatusMsg, TimeMsg, etc.) do not require finite state machines.

When a restore request is not denied, ConfigMgr sets the UnderConfig flag for the server, sends a ConfigServerMsg "Action Initiated" RESPONSE to the Host, and calls RestoreISV operation of NetPM to restore the server to In-Service. ConfigMgr then suspends restore processing and sets up a Server Status PendingQueue entry for the server to become In-Service. Restore processing will not continue until ConfigMgr is informed that the server status is In-Service or the timer expires. When ConfigMgr is informed of the server status change to In-Service, Restore processing is continued by checking the server query status. If the server's query status is DISABLED_SERVER_OOS and the number of active servers is less than the processor service group active server count, ConfigMgr calls EnableQuery operation of NetPM to enable the server's query status and sets the current query status to Pending. ConfigMgr then sends server status messages to the host informing about server and query status change. A QueryStatus PendigQueue entry is set up for the server's query status to become Enabled. Processing is then suspended until the query status becomes enabled or the timer expires. When ConfigMgr is informed of the query status change to Enabled, Restore processing is continued with the sending of server status messages and clearing of the under configuration flag for the server.

Restore fail processing is initiated if the timer expires before the server status changes to In-Service or the requested server information for the other applications is never received. Fail processing involves gracefully downgrading the server to OS-MIN, issuing an IPR, and clearing the under configuration flag for the server. If the timer expires before the query status changes to Enabled, Restore processing is continued with setting the Query Status to Disabled, gracefully downgrading the server to OS-MIN, sending server status messages, issuing an IPR, and clearing the under configuration flag for the server.

When a Graceful Halt request is not denied, ConfigMgr sets the UnderConfig flag for the server, sends a ConfigServerMsg "Action Initiated" RESPONSE to the Host, and calls GraceHalt operation of NetPM to halt the server. If the node is not already halted, ConfigMgr then suspends halt processing and sets up a Server Status Pending Queue entry for the server to become Halted. It then makes an entry to network configuration report indicating a halt was requested by the host. halt processing will not continue until the ConfigMgr is informed that the server status is Halted or the timer expires. When ConfigMgr is informed of the server status change to a halted state, halt processing is continued with the sending of server status messages and clearing of

the under configuration flag for the server. If the timer expires before the server status changes to Halted, Halt fail processing is initiated. Fail processing involves issuing an IPR and clearing the under configuration flag for the server.

When an Immediate Halt request is not denied, ConfigMgr sets the UnderConfig flag for the server, removes all pending server status changes for this server from the status pending queue, and calls ImmedHalt operation of NetPM to halt the server. If the node is not already halted, ConfigMgr suspends halt processing and sets up a Server Status Pending-Queue entry for the server to become Halted. It then makes an entry to the network configuration report indicating a halt was requested by the Host. Halt processing will not continue until the ConfigMgr is informed that the server status is Halted or the timer expires. When ConfigMgr is informed of the server status change to a halted state (or the node is already halted when the halt was issued), halt processing is continued with the sending of server status messages, sending of a ConfigServerMsg "Successfully Completed" RESPONSE to the Host, and clearing of the under configuration flag for the server. If the timer expires before the server status changes to Halted, Halt fail processing is initiated. Fail processing involves issuing an IPR, sending a ConfigServerMsg "Action Failed" RESPONSE to the Host, and clearing the under configuration flag for the server.

When a Graceful Downgrade request is not denied, ConfigMgr sets the UnderConfig flag for the server, sends a ConfigServerMsg "Action Initiated" RESPONSE to the Host, and calls GraceDown operation of NetPM to downgrade the server. If the node is not already at the desired downgraded state, ConfigMgr then suspends downgrade processing and sets up a Server Status PendingQueue entry for the server to become OS-MIN. It then makes an entry to network configuration report indicating a downgrade was requested by the Host. Downgrade processing will not continue until ConfigMgr is informed that the server status is OS-MIN or the timer expires. When ConfigMgr is informed of the server status change to a OS-MIN state (or the node was already at that state), downgrade processing is continued with the sending of server status messages and clearing of the under configuration flag for the server. If the timer expires before the server status changes to a OS-Min state, downgrade fail processing is initiated. Fail processing involves issuing an IPR and clearing the under configuration flag for the server.

When an Immediate Downgrade request is not denied, ConfigMgr sets the UnderConfig flag for the server and calls ImmedDown operation of NetPM to downgrade the server. If the node is not already at the desired downgraded state, ConfigMgr then suspends downgrade processing and sets up a Server Status Pending Queue entry for the server to become OS-MIN. It then makes an entry to network configuration report indicating a downgrade was requested by the Host. Downgrade processing will not continue until ConfigMgr is informed that the server status is OS-MIN or the timer expires. When ConfigMgr is informed of the server status change to a to OS-MIN state (or the node was already at that state), downgrade processing is continued with the sending of server status messages, sending of a ConfigServerMsg "Successfully Completed" RESPONSE to the Host, and clearing of the under configuration Flag for the server.

If the timer expires before the status changes to a OS-MIN state, downgrade fail processing is initiated. Failure processing involves issuing an IPR, sending a ConfigServerMsg "Action Failed" Response to the Host, and clearing the under configuration flag for the server.

When a Graceful or Immediate Boot request is not denied, ConfigMgr sets the UnderConfig flag for the server and sends a ConfigServerMsg "Action Initiated" RESPONSE to the Host. ConfigMgr checks the server status for the server and calls GraceHalt or ImmedHalt operation of NetPM if the server is not at a halted state. If a halt operation is called, processing is suspended until ConfigMgr is informed that the server status is halted or the timer expires. It then makes an entry to network configuration report indicating a boot was requested by the Host.

When ConfigMgr is informed of the server status change to a OS_MIN state (or the node was already at that state), downgrade processing is continued with the sending of server status messages, sending of a ConfigServerMsg "Successfully Completed" RESPONSE to the Host, and clearing of the under configuration flag for the server. If the timer expires before the server status changes to a OS-MIN state, downgrade fail processing is initiated. Fail processing involves issuing an IPR, sending a ConfigServerMsg "Action Failed" RESPONSE to the Host, and clearing the under configuration flag for the server.

When a Graceful or Immediate Boot request is not denied, ConfigMgr sets the UnderConfig flag for the server and sends a ConfigServerMsg "Action Initiated" RESPONSE to the Host. ConfigMgr checks the server status for the server and calls GraceHalt or ImmedHalt operation of NetPM if the server is not at a halted state. If a halt operation is called, processing is suspended until ConfigMgr is informed that the server status is halted or the timer expires. It then makes an entry to network configuration report indicating a boot was requested by the host.

When ConfigMgr has determined that the server is halted, it sends a ResetServerMsg REQUEST to the Host. ConfigMgr creates a Host Response PendingQueue entry to await the ResetServerMsg RESPONSE from the host. Processing is then suspended until the RESPONSE is received or the timer expires. Once the RESPONSE is received, ConfigMgr sets up a ServerStatus Pending Queue entry to await the server status becoming OS-MIN. If the RESPONSE from the Host is not received before the timer expires, an IPR is issued and the under configuration flag is cleared. Once the Server Status becomes OS-MIN, ConfigMgr sends Server status messages to the Host indicating the new server status and clears the under configuration flag. If the timer expires before the server status becomes OS-MIN, ConfigMgr issues an IPR and clears the under configuration flag.

When a Host Resync request is not denied, ConfigMgr determines if the server status is XOX_MIN. If so, SetServerStatus operation of NetPM is called to set the server status to the appropriate Auto/Manual OS_MIN state, server status messages are sent to indicate the new server status, and a ConfigServerMsg "Successful" RESPONSE is sent to the Host. If the server status is not XOS_MIN, an IPR is issued and a ConfigServerMsg "Action Failed" RESPONSE is sent to the Host.

The Application Status Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. Upon receiving an ApplStatusMsg REQUEST type messages from the Host, ConfigMgr determines the application query status and sends a ApplStatusMsg S_REPORT back to the Host with the current application query status. ConfigMgr sends ApplStatusMsg U_REPORT type messages to the Host when server status changes occur or as required during processing of a Host configure server request.

ConfigMgr receives an ASPEC Data REQUEST message from the Host for each Application in the ApplsInfo.des

descriptor file. ConfigMgr queries NetPM to retrieve the information for that application from the NetMAP. A response message containing the ASPEC Data is sent back to the Host, along with a response code indicating success or failure. IPRs will be issued if there is an invalid Application Id, a message other than the ASPEC Data REQUEST message, or a message type other than request.

The Processor service group Info Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. Upon receiving a PSGInfoMsg REQUEST type messages from the Host, ConfigMgr determines the Processor service group Info and sends a PSGInfoMsg S_REPORT back to the Host with the Processor service group information.

The Processor service group Status Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. Upon receiving PSGStatusMsg REQUEST type messages from the Host, ConfigMgr determines the Processor service group query status and sends a PSGStatusMsg S_REPORT back to the Host with the current Processor service group query status. ConfigMgr sends PSGStatusMsg U_REPORT type messages to the Host when server status changes occur or as required during processing of a Host configure server request.

The Query Process Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. ConfigMgr receives QueryProcMsg DISABLE_SERVER, DISABLE_SERVER_FORCED, and ENABLE_SERVER request types from the Host. Upon processing this message, ConfigMgr initiates the enabling/disabling of query processing for the target server by calling the EnableServer/DisableServer operation from NetPM. ConfigMgr will set up a QueryStatus PendingQueue entry for the server and suspend further processing until the query status for the server changes to the desired state or the timer expires. NetPM informs ConfigMgr of a change in query status by calling the NtfyQryStatChange operation of ConfigMgr. When ConfigMgr processes this operation, it will check the QueryStatus Pending Queue entries for the server query status state. If there is an entry with the desired query status, the appropriate success query processing procedure is called to resume processing of the QueryProcMsg. Success processing for the QueryProcMsg involves sending a QueryProcMsg RESPONSE back to the Host indicating the request was successful and changing the active status if necessary for a platform manager node.

If the timer expires before the server query status is in the desired state, the appropriate fail query processing procedure is called to resume processing of the QueryProcMsg. Fail processing for the QueryProcMsg involves issuing an IPR and sending a QueryProcMsg RESPONSE back to the Host indicating the request failed.

The ConfigMgr sends ResetServerMsg REQUEST type messages during boot processing of a server. When the Host requests a boot for a non-PM server, the ResetServerMsg REQUEST is sent after the target server has been halted. ConfigMgr then suspends boot processing and sets up a Host Response Pending Queue entry for a ResetServerMsg RESPONSE type message. Boot processing will not continue until the RESPONSE is received or the timer expires. When ConfigMgr receives the ResetServerMsg RESPONSE type message from the Host, ConfigMgr will check if there is an entry for the ResetServerMsg RESPONSE in the Host Response Pending Queue entry for a RestServerMsg RESPONSE in the Host Response Pending Queue. If so, the appropriate procedure will be called to complete boot processing.

The Routing Info Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the message will be discarded. Upon receiving a RoutingInfoMsg REQUEST type messages from the Host, ConfigMgr sends a RoutingInfoMsg RESPONSE back to the Host indicating the request was acknowledged and attempts to retrieve the Routing Info. Once the Routing info is retrieved, ConfigMgr sends a RoutingInfoMsg S_REPORT back to the Host with the routing information. ConfigMgr sends RoutingInfoMsg U_REPORT type messages to the Host upon request by another subsystem to send routing information. Upon receiving a request to send routing information from another subsystem, ConfigMgr checks the routing pending queue to determine if the Host requested the information. If so, ConfigMgr sends a RoutingInfoMgr S_REPORT to the Host with the routing information. Otherwise, ConfigMgr sends a RoutingInfoMsg U_REPORT to the Host with the routing information. After ConfigMgr sends a U_REPORT to the Host, ConfigMgr waits for the Host to acknowledge receiving the data by sending a RoutingInfoMsg ACK RESPONSE. If no response is received by ConfigMgr within the time limit, ConfigMgr requests the appropriate subsystem to send the application routing information again (to cause a resend of the data to the Host). If a NAK RESPONSE is received from the Host, ConfigMgr issues an IPR indicating a failed response code from the Host.

The Scheduled Action Control Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. When SchedActCtlMsg SET type messages are received from the Host, ConfigMgr calls SetSchedAction operation of NetPM to enable/disable the scheduled actions (such as constant monitoring and generic audits) as desired. ConfigMgr sends a SchedActCtlMsg RESPONSE type back to the Host to indicate whether the Set was successful or not. ConfigMgr has a GetSchedActions operation that can be used by a client to get the Host time information. When this operation is invoked, ConfigMgr sends a SchedActCtlMsg REQUEST type message to the Host. ConfigMgr then sets up a Host Response Pending Queue entry for the desired SchedActCtlMsg S_REPORT from the Host. Processing (of GetSchedActions) is then suspended until the S_REPORT is received or the timer expires. No action is taken if the timer expires before receiving the scheduled actions. When ConfigMgr receives the SchedActCtlMsg S_REPORT type message from the Host, ConfigMgr will check if there is an entry for the SchedActCtlMsg S_REPORT in the Host Response Pending Queue. If so, ConfigMgr calls SetSchedAction operation of NetPM to enable/disable the scheduled actions as desired.

The Server Info Message is precessed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. ConfigMgr sends ServeInfoMsg REQUEST and REQUEST ALL tupe messages to the Host during initialization processing and restore processing of aplatform managerserver. After the message is sent, ConfigMgr suspends processing of the task and sets up a Host Response Pending Queue entry for a

ServerInfoMsg S_REPORT type (and/or COMPLETE type if REQUEST ALL is used). Initialization and restore processing is not continued until the required Server Info is obtained or the timer expires. If the timer expires (before info is obtained) during initialization, ConfigMgr sends the ServerInfoMsg REQUEST or REQUEST ALL again until the information is obtained. If the timer expires (before info is obtained) during restoral of aplatform managerserver, ConfigMgr issues an IPR that the restoral failed.

When ServerInfoMsg S_REPORT and COMPLETE messages are received from the Host, ConfigMgr checks if there is an entry for the ServerInfoMsg S_REPORT or COMPLETE in the Host Response Pending Queue. If so, the appropriate procedure will be called to complete initialization or restore processing. When ServerInfoMsg CHANGE type messages are received from the Host, ConfigMgr determines if it is in an appropriate state to process a server info CHANGE. If so, ConfigMgr informs NetPM of changed server information and sends a ServerInfoMsg RESPONSE type back to the Host to indicate whether the server information was changed successfully or not.

The Server Status Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. Upon receiving a ServerStatusMsg REQUEST type messages from the Host, ConfigMgr obtains the server and query status information and sends a ServerStatusMsg S_REPORT back to the Host with the current status information. ConfigMgr sends ServerStatusMsg U_REPORT type messages to the Host when server status changes occur or as required during processing of a Host configure server request.

The Test Server Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. If the target server is myself and my mateplatform manageris not halted, this platform manager node will discard the request while the otherplatform managerprocesses message. Upon receiving a TestServerMsg REQUEST or ABORT type message from the Host on the MAINT logical link, ConfigMgr determines if the target server's status is MOS_MIN. If so, ConfigMgr sends a TestServerMsg Acknowledge RESPONSE back to the Host. In the future, ConfigMgr will initiate or abort the appropriate test based on whether a REQUEST or ABORT is received. If the target server is not MOS_MIN, ConfigMgr sends a TestServerMsg Server Not MOS-MIN RESPONSE back to the Host. If the target server status cannot be obtained, ConfigMgr sends a TestServerMsg Denied RESPONSE back to the Host and issues an appropriate IPR.

The Time Message is processed by the platform manager node that is In-Service Active. If neither platform manager node is In-Service, the platform manager node that is OS-MIN Active will process the request. Upon receiving a TimeMsg SET type messages from the Host, ConfigMgr calls SetTime operation of NetPM to set the server network time to the appropriate time and sends a TimeMsg RESPONSE back to the host to indicate whether the Set was successful or not. ConfigMgr has a GetTime operation that can be used by a client to get the Host time information. When this operation is invoked, ConfigMgr sends a TimeMsg REQUEST type message to the Host. ConfigMgr then sets up a Host Response Pending Queue entry for the desired TimeMsg S_REPORT from the Host. Processing is then suspended until the S_REPORT is received or the timer expires. No action is taken if the timer expires before receiving the timer information. Upon receiving a TimeMsg S_REPORT type message from the Host, ConfigMgr will check if there is an entry for the TimeMsg RESPONSE in the Host Response Pending Queue. If so, SetTime operation of NetPM is called to set the server network time.

Node Management Services

Node Platform Manager (NodelPMMMain)

The Node Management subsystem provides process management within a single server node. It is responsible for starting/stopping processes within the server node to maintain specific run-levels. Run-levels supported by Node Management are

HALTED (No software running–not even OS)

MIN-SET (OS+Minimal Required Platform Software)

INSERVIConfigurable element (MIN-SET+Common Software)

Network Management informs Node Management of the desired run-level for a specific node. In the event of a process failure, Node Management evaluates the failure and determines what, if any, recovery action is necessary. Recovery actions include ignoring the failure, autostarting the node to the next lower run-level and back to the current run-level, and system shutdown.

NodePM will ve brought up as part of System start-up procedure for each server node. As part of its initialization, NodePM:

Instantiates the NodeMAP object, and after getting the configuration information on the minimum Configurable elements that need to be configured on each servers, it brings up the server node to a minimal operational state (OS-MIN). From this state the server node is allowed only a minimum set of functionality such as bringing the rest of the processes up. The configuration data provided in each node's NodeMAP determines the capabilities of each server node (server nodes withplatform managercapabilities versus server nodes with query processing capabilities).

Creates the NodePM server object to handle the NetPM requests to perform operations within the same server node.

Per NetPM request, NodePM (through operations provided by its server object) can perform the following operations:

Bring up its server node to a fully operation state (INSERVIConfigurable element) from a minimal operational state (OS-MIN) (RestoreNode operation).

Bring down its server node to a minimal (OS-MIN) or halted (HALT) operational state from a fully operational state (IN-SERVIConfigurable element) (RemoveNode operation).

Enable/Disable the query processing on its server node.

Provide status information on Configurable elements.

NodePM reports any change of status on each IPU autonomously to NetPM (NodePM utilizes the operation provided by NetPM to report the status change).

FIG. **8** is a diagram showing the legal service state transitions for a node. Notice that all automatic states transition to other automatic states and all manual states transition to other manual states. There is no legal transition from a manual state to an automatic state. The ISV state has no automatic or manual designation at this time. States can transition form/to IN-SERVICE (ISV) state **200** to/form any other state. The acronyms used in FIG. **8** are decoded as follows:

ISV 200	in service
OOSAM 202	automatic out of service minimal
OOSMM 204	manual out of service minimal
OOSAN 206	automatic out of service-halted
OOSMN 208	manual out of service-halted
ABOOT 210	automatic boot
MBOOT 212	manual boot
ADOWN 214	automatic downgrade
MDOWN 216	manual downgrade
AHALT 218	automatic halt
MHALT 220	manual halt
AREST 222	automatic restore
MREST 224	manual restore

Node System Integrity (NodeSIMain)

The Node System Integrity subsystem (class name NodeSI) provides fault isolation and monitoring services within a single server node. All process failures are logged by this subsystem and forwarded to node Management for recovery action. Node System Integrity implements the following features:

Passive process monitoring (signal catching)

Inter-nodal communications monitoring

Local fault reporting

The System Integrity (SI) capabilities of the AIN platform can be categorized as those providing capabilities across the server nodes of the platform, and those that provide capabilities within a single server node. While NetSI handles the system integrity capabilities at the platform level, the NodeSI provides system integrity at the single node level. NodeSI resides in every server node of the platform, and provides operations through which processes for each configurable element can report fault conditions on that process. These faults include:

Faults detected by Constant Monitor object on each process.

Inter nodal communication failures.

Communication failures between the host and server network.

Faults detected by IM Server process.

It also performs node constant monitoring of all connections to/from the node. It a communication fault is detected, NodeSI will inform NetSI of the communication fault. Depending on the reported fault, NodeSI will take appropriate actions, including issuing IPRs, and downgrading the node's state (in cooperation with the NodePM).

NodeSI monitors the disk utilization on each server node, the issues appropriate IPR when the total capacity used on a particular file system exceeds a certain threshold. NodeSI communication with other objects is handled via the DOME interface. NodeSI gets the list of all IPUs in the configuration from NodePM. An array is set up containing the following information from each IPU:

IPU information received from NodePM

IPU status

Fault count

Alive message received indicator

An array index into this list is used to communicate status with the other NodeSI's rather than the node name since string comparisons con be costly in terms of speed and efficiency. Therefore, it is important that each node in the configuration have the same IPU list in the same order.

NodeSI registers with NodePM to get node state notifications. When NodeSI is informed of a status change for another IPU, it will update the IPU status in the IPU array. It the status change is to the halted state, NodeSI will clear the fault counts and alive message received indicator.

NodeSI has two timers to handle its constant monitoring function:

BroadcastTimer—timer that causes NodeSI to broadcast "I'm alive" messages to the other NodeSI's in its view.

ConMonChkTimer—timer that causes NodeSI to determine if the appropriate "I'm alive" messages have been received for all connections within the time interval.

When NodeSI is informed that is node is OS-MIN, it starts broadcasting "I'm alive" messages to the other NodeSI's in its view. It then triggers the BroadcastTimer. Upon BroadcastTimer expiration, NodeSI immediately rebroadcasts the "I'm alive" messages and retrigger the BroadcastTimer. This will interrupt any NodeSI processing that may be going on.

When NodeSI receives an "I'm alive" message from another NodeSI, it marks the appropriate IPU array entry's Alive message received indicator.

When NodeSI is informed that is node is OS-MIN, it triggers the ConMonChkTimer. Upon ConMonChkTimer expiration, NodeSI makes a Dome call to the CommFailCheck operation to perform communication failure checking and retrigger the timer. It is using the DOME call to itself in order to assure that priority is given to broadcasting the alive messages.

Communication failure processing involves checking each IPU in its array to determine if an alive message have been received since the last time it checked. If so, the Alive message received indicator is cleared. If no message has been received and the IPU status is not halted, the fault count for that node will be incremented. If the number of faults for that IPU is at its maximum, NodeSI reports a communication failure to NetSI.

The maximum number of fault counts is a configurable value that can be read in from the command line by using the keyword "MAX_COMM_FAULTS". If no value is given, the default number of fault counts will be 2. Also, if the value given in the command line is less than 2, the maximum number will be set to 2.

The number of seconds between each broadcast of alive messages is a configurable value that can be read in from the command line using the keyword "BRDCAST_ALIVE-SECS". If no value is given, the default number of seconds between broadcasts will be 1 second. If the value given in the command line is less than 1 second, the number of seconds will be set to 1.

The number of seconds between each constant monitoring check is a configurable value that cen be read in from the command line using the keyword "CONMON_CHK_SECS". If no value is given, the default number of seconds between checks will be 2 seconds. If the value given in the command line is less than 2 seconds, the number of seconds will be set to 2.

NodeSI is started by NodePM as part of every node's start-up, and prior to other processes start-up. As part of its initialization, NodeSI reads a descriptor file (Fault.des) containing the definition of the faults detected by the NodeSI, and creates a list (FaultInfoList) of those fault records. Each fault record (FaultInfo) contains the following parts:

FaultId—Fault Identification.

FaultActId—Action to be taken per Fault reported.

As faults are received, NodeSI will search for the fault record in its list (FaultInfoList) using the fault's Id, and performs the action associated with that fault. These actions may include:

Issuing appropriate IPRs.

Halting the node in case of detecting catastrophic faults on NodePM process.

Reporting autonomous status changes on Configurable elements to NodePM.

Reporting communication failures to NodePM and in turn to NetSI.

All faults (originated from Constant Monitor or other processes) will be reported to the NodeSI by each process via NotifyFault( ) operation of NodeSI. NodeSI keeps track of disk utilization on the server node, and issues an IPR if 80 was used.

NodePM Interface

NodeSI uses the interface provided by NodePM to report the autonomous changes in a Configurable element's status (AutoChgCEStat(. . .)). Depending on the configurable element's impact on the state of the node, the status change may cause NodePM to perform any of the following actions:

Downgrade Node's State—This action is performed if the configurable element's status change had a major impact on the current operational state of the node. Prior to doing this, NodePM will inform the NetSI of its intent, and starts a timer. Then upon request from NetPM or time-out, it will downgrade the node's state.

Report Communication Failure—This action is performed if the configurable element's status change indicated an internodal communication failure (TCP link goes out of service). For this situation, NodePM will notify NetSI of communication failure, and attempts to establish the communications again.

NetSI Interface

NetSI provides operations, used by NodeSI and/or NodePM to report the following conditions:

Autonomous changes in an IPU's status (DowngradeIPStat(. . .))—In this situation, NetSI downgrades the node through NetPM (requests NetPM to downgrade, if the node was not halted already).

Communication failures (CommFaultRprt(. . .))—In this situation, if communications failure to the same IPU was reported by other IPUs, then NetSI will mark that IPU as the IPU in fault, and attempts to downgrade it through NetPM.

Constant Monitor Interface

Each Configurable element process is required to instantiate the Constant Monitor object, in order to detect and report abnormal conditions/events generating different signals on the process. Constant Monitor reports these conditions via NotifyFault( ) operation of NodeSI. In case of failure to communicate the fault to NodeSI, the Constant Monitor may HALT the node, depending on the options set at the time of its instantiation.

Message Handler/Logical Links Interface

Message Handler or Logical Link configurable element processes utilize the NodeSI operation NotifyFault( ), to report faults on DNI/TCP links.

Service Manager (SMProcess)

The service management subsystem provides process control for application processes. Application processes are only run after the node has achieved the IN SERVICE run-level. Application processes can be individually removed/restored and enabled/disabled on a server node. Network management informs service management as to which applications to remove, restore, enable, disable. Features implemented by service management include:

Active Process Monitoring (Heartbeats, Audits)

Multiple process instance support

Application Process State Management
- Administrative State
- Operational State
- Usage State

Application process state change notification

For the telecom platform Navigator feature to present a consistent configurable element interface, a change has been made to have service management start System configurable elements instead of NodePM. By doing this, all processes in the system (except service management) are started by service management, so the features of a configurable element are now the same system-wide. To create a telecom platform Navigator GUI, a consistent view of a telecom platform system has to exist. FIG. **9A** is a diagram that shows the new relationship that exists during node initialization between entities in the telecom platform. For a configurable element to be able to take advantage of all service management functionality, the service management interface needs to be followed.

A boot script **230** is created to be the first thing to run on all nodes. When the boot program **230** runs, it will identify the platform manager node **232**, and copy the active platform manager node's Tcl descriptor file **234** over to use to bring up that node. If is determines that it is the first platform manager node to come up, it will use the existing Tcl descriptor file **234** to run.

The platform manager subsystem, and the service management subsystem **236** have a different concept of what a configurable element **238** is in the previous version of the platform. These two concepts are joined into one configurable element concept, merging their separate functionalities. To do this, the platform manager subsystem will no longer remove and restore configurable elements, but will inform service management when it wants a configurable element to be removed and restored. Service management will now be the first telecom platform program started, and will always start NodePM as part of its initialization. NodePM will then be in control of starting and stopping processes that same as it was before, only through the service management, not through the old RemoveCE and RestoreCE functionality.

FIG. **9B** is a message flow diagram showing node initialization into the MIN_SET state. FIG. **9C** is a message flow diagram showing node initialization into the IN_SEVICE state, and FIG. **9D** is a message flow diagram showing node initialization into the POST_ISV state.

FIG. **10** outlines the messages protocol that is used between SM and a Configurable element. If a configurable element cannot for link a service management interface (SMI) object into it, service management can still start that configurable element, but many of the features that service management provides will not be available.

Event Manager (eventmanagerimpl)

The event manager subsystem provides the ability for a users to generically issue event notification to one or more registered parties. Multiple Event::Manager object instances may exist in the system. A node level Event::Manager exists on all nodes. Other Event::Manager instances may also exist to provide the ability for interested parties to register for events that are special to a process. The eventmanagerimpl program provides an Event::Manager object instance for the mode that it is running on. Events that are relevant to a node get issued through that Event::Manager instance. Users

interested in events on a particular node can bind to that nodes Event::Manager instance by using that nodes name as the Event::Manager name. Programs can also embed an Event::Manager object within their program. The IprMgrImpl program is an example of a program that does this. The IprMgrImpl has an Event::Manager named IprEventMgr. Users that wish to receive IPR events. Users that are interested in a particular event may register with a particular Event::Manager instance to receive that event through that Event::Manager instance. The Event::Manager does not persistently store the list of registered parties. If the Event::Manager tries to forward an event to a Event::Receiver that has gone away, that Event::Receiver is removed form the list.

FIG. **11** shows two examples of uses for Event::Manager **250** in the telecom platform system. The eventmanagerimpl **252** contains the node Event::Manager object instance **250**. The NodePMMain telecom platform program **254** uses this Event::Manager **250** to issue an event when the node changes state. The application program **256** then creates an Event::Receiver object **268** and passed a CORBA object reference to the register call on the "Node123" Event::Manager **250**, When NodePMMain **254** generates an event by calling notify on the "Node123" Event::Manager **250**, that Event:Manager **250** will find all of the Event::Receiver objects **258** that have registered to receive this event. Seeing that the application program has registered for this event, the Event::Manager **250** will call the notify( ) method on that Event::Receiver object **258** which will cause the notify ( ) method to be invoked in the Application program **256**. In the example above, the Application program **256** has also registered with the "IprEventMgr" Event::Manager **260** in the IprMgrImpl program **262**. When NodePMMMain **254** uses the IprMgrImpl interface to issue an IPR, the IprMgrImpl program **262** does the lookup on that IPR and performs verification, and calls notify ( ) on the "IprEventMgr" Event::Manager **260**. This cause that Event::Manager **250** to forward the generated event to the Event::Receiver **264** in the application program **256** that was passed in the register call.

Application programs **256** can create their own Event::Manager with its own name the same way the IprMgrImpl program did. Event::Manager instances need to have unique names in the system to prevent generating an event to the incorrect Event::Manager, or to help isolate a user from registering with the incorrect Event::Manager.

IPR/ALARM Services

The Information and Problem Reporting (IPR) subsystem provides all processes in the system with the ability to issue Information and Problem Reports. IPRs are the standard mechanism used to inform users of the system about error conditions or other pertinent system information. The Information and Problem Reporting subsystem implements the collection of IPRs in the telecom platform. An alarm is a mechanism which may be attached to an IPR. Alarm services are not available now, but will be available in future release of telecom platform.

The IPR subsystem provides several features. It provides active/standby IPR service redundancy, the ability to forward IPRs to registered receivers, the ability to forward IPRs to the host, the ability to display IPRs in real-time, backward compatibility with the legacy PAConfigurable element IPR interface, a CORBA IPR interface, the ability to use an IPR dictionary to validate IPRs, the ability to provide additional information about the IPR that was issued from the IPR dictionary, and the ability to provision IPR in the IPR dictionary.

Referring to FIG. **12**, the IprMgrImpl program is the collection point for all IPRs in a telecom platform site. This program contains the IprMgrImpl CORBA server object. The IprMgrImpl object runs on each of the active/standby platform manager nodes. The active/standby state that the IprMgrImpl reacts to is the node level active/standby state of the telecom platform manager nodes. The standby IprMgrImpl object will unpublish its interface, and the active IprMgrImpl object will publish its CORBA interface when the platform manager nodes change active/standby state. By doing this, client users of both the IprMgr and IPRClient interfaces will have their IPRs forwarded to the active IprMgrImpl object.

The Event Manager subsystem is used within the IPR subsystem to distribute IPRs. This allows IPRs to be forwarded to multiple destinations. By using the Event Manager, additional IPR features can be easily added to the system without incurring interface changes. The Event Manager mechanism of the IPR subsystem is currently used within the telecom platform to provide some existing IPR services. The real-time IPR_GUI **270** registers to receive IPRs for the purpose of displaying IPRs as they occur. The Ipr2host program **272** registers with the IPR subsystem to receive IPRs and forwards them to the host. An IPR logger may also register to receive IPRs to log to disk.

The ipr2host program **272** is responsible for forwarding IPRs to the host. It receives IPRs from the IprMgrImpl's Event Manager, and formats a host message to forward on. All IPRs that get forwarded to the host use the message handler subsystem to forward IPRs over the IPR_ASSERT logical link.

The IPR subsystem has a two external interfaces: the IPRClient interface **274** and the CORBA IPR interface **276**. The IPRClient interface **276** exists for backward compatibility with previous PAConfigurable element releases. Once the issued IPR from the IPRClient interface **274** has been converted by the IPRClient code, an IPR is issued using the IprMgrImpl CORBA interface to route the IPR to the active IprMgrImpl object. This interface still uses the LOCIPRDB.DSK IPR dictionary as input for converting the old PAConfigurable element IPRs to the current IPR subsystem format. This requires that a LOCIPRDB.DSK reside on each node that has programs that issue IPRs. The LOCIPRDB.DSK dictionary was used in the previous releases to do IPR verification before IPRs were forwarded to the host. The RegisterIPR utility is used to enter IPRs into the LOCIPRDB.DSK dictionary. The fields in the database entries include: ASCII key (IPR text), host IPR number, IPR priority, number of data words used, and data word format. In order to test the IPRMgr, IPRs must be defined in ipr.in which will be converted to a keyed dictionary (via the RegisterIPR utility).

The IprMgrImpl interface is a CORBA IDL interface. If an IPR is issued using this interface, it is not required to be entered in the LOCIPRDB.DSK dictionary. When the IprMgrImpl object receives an issued IPR, it looks it up in its IPR dictionary and constructs an IPR event to be issued. The IPR event contains information that was passed from the client that issued the IPR, and information from the IPR dictionary. IPRs must be added to the IPR dictionary and the MegaHub host IPR dictionaries prior to issuance of an IPRs. The IprDriver tool is used to add IPRs to the IprMgrImpl IPR dictionary. The reformat and reformat2 scripts exists to assist in converting a VAX IPR file to a format that can be used with the IprDriver to populate the IprMgrImpl IPR dictionary.

FIG. **13** illustrates the scenario where an application issues an IPR, the IPR Manager processes it, and the Event Manager routes the IPR to an IPR GUI for visual display.

1) The IPR GUI registers an interest in receiving all IPRs reported to the IPR Event Manager.

2) An application issues an IPR.

3) The IPR Manager forwards the IPR to the Event Manager.

4) The Event Manager distributes the IPR to the IPR GUI.

FIG. **14** is an example of an IPR View GUI screen print. The IPR View GUI application provides the display of IPRs in a split window. In the top pane a graphical view of IPRs is shown with costs vs. time displayed on category basis. The bottom pane displays a traditional full/brief text view of IPRs. Subcategories may be viewed and a number of customizations of the display are allowed. In addition, filtering and highlighting are available for the IPRs displayed. Communication is handled via CORBA.

Statistics Services

Data Collection (DcMProcess, DcProcess)

Referring to FIG. **15**, the data collection subsystem (DC) **298** provides the traffic measuring functionality for the application programs within a node. These measurements are counts recorded by the PegCounter class and elapsed time recorded by the TimeMeter class. PegCounter **299** testing will indirectly test shared memory **300** and semaphores. Client processes **301** peg to shared memory **300**, and data collection **298** collects from shared memory **300** and sends to DCMaster **302**. Every 30 minutes, data collection **298** sends the DCMaster **302** (in the active platform manager node) the 30 minutes worth of peg counter slots **299** and then data collection zeros out those slots. The active platform manager node **304** updates the standby platform manager node **306**.

Referring to FIG. **16**, the statistic services or data collection subsystem **320** provides the traffic metering and measurement capabilities of the platform. This subsystem **320** supports the creation, collection, and reporting of statistical measures like peg counters, time meters, threshold counters, collection and querying. PegCounters **322** and TimeMeters **324** are shown supported across a distributed application. Features implemented by the data collection subsystem **320** include:

PegCounter **322** and TimeMeter **324** API Support

Collection of accumulated data from multiple nodes

Reporting GUI for local viewing of statistics

User defined measurement sets for report customizing

Threshold Counters (TCServer)

The threshold counter subsystem may be implemented as an object request broker (ORB) distributed object, using the orbeline ORB implementation. Applications are connected via Orbeline to a server object resident in the platform manager nodes. The server reports counter threshold crossings to applications via distributed object messaging environment (DOME). The server object are created by the thresholds counter server process, TCServer. Each TCServer process also communicates via Orbeline with the TCServers on remote nodes so that counters can be synchronized across sites. The TCServer keeps all counters in persistent storage using the persistent dictionary supplied in the common services library as template class RepShmDict.

FIG. **17** shows the communication paths between application processes **340** and the counter server processes. The TCServer process **342** communicates with application processes **340** via both Orbeline **344** and DOME **346**. The TCServer process **342** runs in an orbeline impl_is_ready loop, waiting for service requests from either application processes **340** or from a TCServer process **342** on another node. It makes a DOME ReqServ call to notify application processes **340** that a counter has reached its threshold.

Referring to FIG. **18**, the threshold counter subsystem **360** API hides the orbeline-specific portions of the implementation from the application programmer. Instead, the client side of the subsystem will consist of two layers: an ORB-independent layer **362**, and an orbeline-dependent layer **364**. Although the orbeline-specific implementation of the subsystem is hidden from the application programmer, the distributed nature of the subsystem is not. To minimize the time required for counter increments, counter increments are buffered in the API, and sent to the server in batches. This means that the application is unable to receive immediate notification of the success or failure of some operations on the API objects.

Communications Services

Message Handling (MsgHndl, LinkXXX)

As shown in FIGS. **19** and **20**, the Message Handling subsystem **370** provides message based interprocessor communications services. Generally all interprocess communication between processes on the server nodes is carried out via the Distributed Object Messaging Environment (DOME) **372** shown in FIG. **21**. DOME **372** uses the Message Handling subsystem **370** when information must be communicated across node boundaries. The Message Handling subsystem **370** is also used for communication to non-server external systems such as the SCP Host. The Message Handling subsystem **370** implements the following features.

Common interface for multiple protocols.

TCP/IP **374**

UDP/IP **376**

DECNET **378**

Single access identifier (Logical Link Group Name) for multiple links with same destination.

Redundant link management (improves scalability)

Link failure recovery

Asynchronous receive interface

Distributed Object Services

Referring to FIG. **21**, DOME **372** is a client/server interface used for interprocess client/server communication. It contains server interfaces **382** which allow server processes **382** to register objects and member functions for use by client processes **384**. DOME **372** contains a shared memory database **380** to store the server descriptions and a stand-alone DOMEServices process (domeSrv) which maintains the server object descriptions from other nodes. It also contains client interfaces **384** which provide access to any registered server object in the node's DOME database.

The Interprocess Communications subsystem consists mainly of DOME. DOME provides the ability for a process to register a server object and it's methods in a way that allows other processes in the system to invoke those methods. DOME supports various modes of registration and access including many special routing options that aid in the development of fault resilient software. Features implemented by the Interprocess Communications subsystem include:

Registered Object Name Management across nodes and sites

Prioritized request handling

Active/Standby Object request routing

Load Shared Object request routing

Broadcast Object request routing

Blocking/Non-Blocking Object requests

Common Services

The Common Utilities subsystem provides a library of programming tools to aid in the rapid development of

processes designed to run on or within the platform layer. The features implemented by the Common Utilities subsystem include:

Command Line Object

Trace Object

Shared Memory Object

Semaphore Object

Keyed Dictionary Object

List Object

Replicated Keyed Dictionary Object

Shared Memory Dictionary Object

etc.

DbgTrace Object

Referring to FIG. **22**, the DbgTrace facilities **400** provides the ability to issue trace messages to a trace buffer, to a file, and/or to standard error. Trace data can be entered in two different formats: standard print format, and a data buffer dump format. A mask **402** may be used to filter out different levels of messages. There are 32 possible mask levels for each DbgTrace group.

The DbgCntl interface **404** is the control interface for DbgTrace objects **400**. It allows users to specify many different aspects of the DbgTrace facility **400**. This interface allows users to do the following things on DbgTrace objects **400**:

Set/Get the mask **402** for a DbgTrace group **400**.

Set/get the size of the internal message buffer **410**.

Get a list of existing groups.

Turn on/off display to standard error.

Turn on/off dumping of traces one at a time to a file.

Enable/disable the ability to dump traces out to file before they get overwritten.

A DbgDisk interface allows users to specify which file the trace buffer **410** will be written to on all write requests.

The DbgTrace facility **400** allows the users to create different DbgTrace objects **400** that can each belong to one of multiple groups. This allows users to have a unique mask value for each group. All traces issued through the DbgTrace interface **400** get stored in an internal message buffer. Users can also specify whether to issue traces to standard error in addition to the internal buffer.

Trace Object

The Trace object provides the user the ability to optionally issue trace messages to standard error. When the user issues a trace, a mask is specified which represents the trace level that this trace will be output for. The Trace interface allows the user to specify a mask which all instances of trace in that UNIX process will use to determine whether or not to issue the trace message. The trace mask may supports eight unique mask values.

Dictionary Management System

Referring to FIG. **23**, Dictionary Management provides classes which are designed to support data storage and access. Dictionaries can be stored on disk (persistent) or stored in memory. Dictionaries can also be private (used by local process only) or shared (accessible by multiple processes). The purposes of these dictionaries are defined by the application program. The primary interaction between DmsMaster **430** and DmsServer **432** is that DmsMaster **430** updates DmsServer **432** when it receives an update message from the application. DmsMaster **430** runs as active/standby in the platform manager nodes, and DmsServer **432** runs in all (or a subset) of the IPUs.

Event Services

Event services provide the capability to generate and distribute specific occurrences significant to a task among loosely coupled processes. An example of an event is the completion of an input/output transfer. The event services may be a CORBA-based interprocess communication facility. It uses standard CORBA requests that result in the execution of an operation by an object. This is accomplished through the event manager implemementation program.

By defining two distinct roles for objects, communication is decoupled between objects; creating asynchronous communication. One object receives and accumulates new events, while the other object registers an interest to be forwarded these new events. This is accomplished by two CORBA classes, EventManager and EventReceiver. EventManager provides an interface definition language (IDL) interface for receiving new events. EventReceiver provides an interface definition language interface for clients interested in receiving events.

Software and Hardware Representation

FIG. **24** shows the hardware view of a telecom platform system. At the highest level, a telecom platform system consists of one or more sites **440**. Within a site **440**, multiple nodes **442** exist.

The software representation is a hierarchy allowing components of software to be grouped together. FIG. **25** shows this hierarchy. An Application **450** exists at the highest level. An Application **450** is made up of one or more configurable element sets **452**, which is made up of one or more configurable elements **454**. Multiple applications **450** can be defined within a system. All of the applications **450** within a system make up the software representation of a system.

The dynamic mapping of software onto hardware representation of a system shown in FIG. **26** depicts how pieces of an application **450** are placed onto nodes **442**. Sites **440** contain applications **450**. Applications **450** have processor service groups **456**. Processor service groups **456** span multiple nodes **442**. Nodes **442** have configurable element sets **452** placed on them. Configurable elements **454** reside within configurable element sets **452**. For example, a software representation of a time dependent routing application may have two configurable element sets: WestCoastSet and EastCoastSet. Within the WestCoastSet, the time dependent routing application could have all of the programs that need to run on the nodes targeted to handle West Coast calls. These might include database programs, link processes, etc. that are configured specifically for West Coast handling. Within the EastCoastSet, the time dependent routing application may have all of the programs that need to run on the nodes targeted to handle West Coast calls. The time dependent routing application would then be allocated onto a site. Nodes that will run the time dependent routing application will be grouped into processor service groups. The configurable element sets for the application would then be placed on nodes that have been placed into a time dependent routing application processor service group.

Although several embodiments of the present invention and its advantages have been described in detail, it should be understood that mutations, changes, substitutions, transformations, modifications, variations, and alterations can be made therein without departing from the teachings of the present invention, the spirit and scope of the invention being set forth by the appended claims.

What is claimed is:

1. A method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node

at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network, comprising:

supplying network management processes operable to provide inter-node configuration, monitoring and management functionality;

supplying node management processes operable to provide node initialization, configuration, monitoring, and management functionality;

supplying event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events;

supplying common processes operable to provide a library of a plurality of programming tools for the development of the application programs;

supplying communications processes operable to provide message handling functionality; and

supplying distributed object processes operable to provide a distributed database repository for object-based communications.

2. The method, as set forth in claim 1, wherein providing the network management processes comprise:

providing a network platform manager operable to remove nodes from service, restore nodes to service, remove applications from service, and restore applications to service;

providing a network system integrity manager operable to monitor the nodes and to enable failed nodes to recover; and

providing a configuration manager operable to interface with a host coupled to the telecom platform.

3. The method, as set forth in claim 2, wherein providing the node management processes comprise:

providing a node platform manager operable to provide management functions for a node;

providing a service manager operable to start and stop processes at the direction of the node platform manager; and

providing a node system integrity manager operable to monitor inter-node links.

4. The method, as set forth in claim 3, comprising:

monitoring and detecting a failure in a configurable element;

notifying the fault to the service manager;

generating, by the service manager, a status change for the configurable element and forwarding the notification to the node system integrity manager;

forwarding, by the node system integrity manager, the notification to the node platform manager;

determining, by the node platform manager, the node status in response to the failed configurable element; and

notifying the net platform manager, by the node platform manger, of a node status change.

5. The method, as set forth in claim 4, further comprising:

determining, by the network platform manager, a status change in an application having the failed configurable element and a status change in a processor service group having the application having the failed configurable element; and

notifying any status change to the configuration manager.

6. The method, as set forth in claim 5, further comprising forwarding, by the configuration manager, a node, processor service group or application status change to a host.

7. The method, as set forth in claim 3, further comprising:

registering with an event manager, by an application, an interest to receive a particular event;

sending, by an event receiver, the particular event to the registered application.

8. The method, as set forth in claim 1, further comprising:

running the network management processes on at least one platform management node; and

running the node management processes on at least one application node coupled to the at least one platform management node.

9. The method, as set forth in claim 8, further comprising running the network management processes and the node management processes on a platform management node also serving as an application node.

10. The method, as set forth in claim 8, further comprising:

operating a first platform management node in an active mode; and

operating a second platform management node in a standby mode.

11. The method, as set forth in claim 8, further comprising operating two or more platform management nodes operating in a load-sharing mode.

12. The method, as set forth in claim 1, further comprising supplying statistics processes operable to provide methods to access system measurement data and to generate reports on the system measurement data.

13. The method, as set forth in claim 12, wherein supplying statistics processes comprise:

providing a peg counter process operable to count specific events occurring across multiple nodes;

providing a time metering process operable to accumulate the duration of a specific event;

providing a data collection process operable to collect counter data on a node and storing the collected data.

14. The method, as set forth in claim 1, further comprising supplying information and problem report and alarm processes operable to provide error condition monitoring, alarms, and reporting.

15. The method, as set forth in claim 1, further comprising supplying dictionary processes operable to provide data storage and access methods.

16. The method, as set forth in claim 1, further comprising supplying graphical user interface processes operable to provide graphical user interface building methods.

17. The method, as set forth in claim 1, wherein providing the event processes comprise:

providing an event manager operable to register client processes wishing to receive events; and

providing an event receiver operable to provide an interface for client processes which are registered to receive events.

18. The method, as set forth in claim 1, wherein providing the common processes comprise providing a timer manager operable to provide date and time functionality.

19. The method, as set forth in claim 1, further comprising:

running a boot script;

starting a service manager in accordance to the boot script;

starting, by the service manager, a node platform manager for a node;

starting, by the service manager, PRE-MIN configuration elements for the node;

starting, by the service manager, OS-MIN configuration elements for the node; and

upgrading a state of the node in response to the OS-MIN configuration elements in the node.

20. A telecom platform forming an interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network, comprising:

network management processes operable to provide inter-node configuration, monitoring and management functionality;

node management processes operable to provide node initialization, configuration, monitoring, and management functionality;

event processes operable to provide initialization, termination, and distribution of tasks in response to predetermined events;

common processes operable to provide a library of a plurality of programming tools for the development of the application programs;

communications processes operable to provide message handling functionality; and

distributed object processes operable to provide a distributed database repository for object-based communications.

21. The telecom platform, as set forth in claim **20**, further comprising:

at least one platform management node on which network management processes are supported;

at least one application node coupled to the at least one platform management node on which node management processes are supported.

22. The telecom platform, as set forth in claim **21**, wherein the at least one platform management node is also the at least one application node.

23. The telecom platform, as set forth in claim **21**, wherein the at least one platform management node comprises:

a first platform management node operating in an active mode; and

a second platform management node operating in a standby mode.

24. The telecom platform, as set forth in claim **21**, wherein the at least one platform management node comprises two or more platform management nodes operating in a load-sharing mode.

25. The telecom platform, as set forth in claim **20**, further comprising statistics processes operable to provide methods to access system measurement data and to generate reports on the system measurement data.

26. The telecom platform, as set forth in claim **25**, wherein the statistics processes comprise:

a peg counter process operable to count specific events occurring across multiple nodes;

a time metering process operable to accumulate the duration of a specific event;

a data collection process operable to collect counter data on a node and storing the collected data.

27. The telecom platform, as set forth in claim **20**, further comprising information and problem report and alarm processes operable to provide error condition monitoring, alarms, and reporting.

28. The telecom platform, as set forth in claim **20**, further comprising dictionary processes operable to provide data storage and access methods.

29. The telecom platform, as set forth in claim **20**, further comprising graphical user interface processes operable to provide graphical user interface building methods.

30. The telecom platform, as set forth in claim **20**, wherein the network management processes comprise:

a network platform manager operable to remove nodes from service, restore nodes to service, remove applications from service, and restore applications to service;

a network system integrity manager operable to monitor the nodes and to enable failed nodes to recover; and

a configuration manager operable to interface with a host coupled to the telecom platform.

31. The telecom platform, as set forth in claim **20**, wherein the node management processes comprise:

a node platform manager operable to provide management functions for a node;

a service manager operable to start and stop processes at the direction of the node platform manager; and

a node system integrity manager operable to monitor inter-node links.

32. The telecom platform, as set forth in claim **20**, wherein the event processes comprise:

an event manager operable to register client processes wishing to receive events; and

an event receiver operable to provide an interface for client processes which are registered to receive events.

33. The telecom platform, as set forth in claim **20**, wherein the common processes comprise a timer manager operable to provide date and time functionality.

34. A method of providing a software interface between application programs performing telecommunications functions and an operating system running on at least one node at a site supporting the application programs, and further forming an interface between the application programs and a telecommunications network, comprising:

providing a network platform manager operable to remove nodes from service, restore nodes to service, remove applications from service, and restore applications to service;

providing a network system integrity manager operable to monitor the nodes and to enable failed nodes to recover;

providing a configuration manager operable to interface with a host coupled to the telecom platform;

providing a node platform manager operable to provide management functions for a node;

providing a service manager operable to start and stop processes at the direction of the node platform manager; and

providing a node system integrity manager operable to monitor inter-node links.

35. The method, as set forth in claim **34**, comprising:

monitoring and detecting a failure in a configurable element;

notifying the fault to the service manager;

generating, by the service manager, a status change for the configurable element and forwarding the notification to the node system integrity manager;

forwarding, by the node system integrity manager, the notification to the node platform manager;

determining, by the node platform manager, the node status in response to the failed configurable element; and

notifying the net platform manager, by the node platform manger, of a node status change.

36. The method, as set forth in claim **35**, further comprising:

determining, by the net platform manager, a status change in an application having the failed configurable element and a status change a processor service group having the application having the failed configurable element; and

notifying any status change to the configuration manager.

37. The method, as set forth in claim **36**, further comprising forwarding, by the configuration manager, a node, processor service group or application status change to a host.

38. The method, as set forth in claim **34**, further comprising:

providing an event manager operable to register client processes wishing to receive events; and

providing an event receiver operable to provide an interface for client processes which are registered to receive events.

39. The method, as set forth in claim **34**, further comprising providing a timer manager operable to provide date and time functionality.

40. The method, as set forth in claim **34**, further comprising:

providing a peg counter process operable to count specific events occurring across multiple nodes;

providing a time metering process operable to accumulate the duration of a specific event;

providing a data collection process operable to collect counter data on a node and storing the collected data.

41. The method, as set forth in claim **34**, further comprising:

running a boot script;

starting a service manager in accordance to the boot script;

starting, by the service manager, a node platform manager for a node;

starting, by the service manager, PRE-MIN configuration elements for the node;

starting, by the service manager, OS-MIN configuration elements for the node; and

upgrading a state of the node in response to the OS-MIN configuration elements in the node.

42. The method, as set forth in claim **34**, further comprising:

registering with an event manager, by an application, an interest to receive a particular event;

sending, by an event receiver, the particular event to the registered application.

* * * * *

US006185590B1

(12) **United States Patent**
Klein

(10) **Patent No.: US 6,185,590 B1**
(45) **Date of Patent: Feb. 6, 2001**

(54) **PROCESS AND ARCHITECTURE FOR USE ON STAND-ALONE MACHINE AND IN DISTRIBUTED COMPUTER ARCHITECTURE FOR CLIENT SERVER AND/OR INTRANET AND/OR INTERNET OPERATING ENVIRONMENTS**

(75) Inventor: **Laurence C. Klein**, Silver Spring, MD (US)

(73) Assignee: **Imagination Software**, Silver Spring, MD (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **08/950,838**

(22) Filed: **Oct. 15, 1997**

Related U.S. Application Data

(63) Continuation-in-part of application No. 08/911,083, filed on Aug. 14, 1997.

(60) Provisional application No. 60/028,129, filed on Oct. 18, 1996, provisional application No. 60/028,522, filed on Oct. 18, 1996, provisional application No. 60/028,128, filed on Oct. 18, 1996, provisional application No. 60/028,697, filed on Oct. 18, 1996, provisional application No. 60/028,639, filed on Oct. 18, 1996, and provisional application No. 60/028,685, filed on Oct. 18, 1996.

(51) **Int. Cl.**[7] **G06F 13/00**; G06F 15/00

(52) **U.S. Cl.** **707/526**; 345/329

(58) **Field of Search** 707/526, 513; 395/500, 701, 712, 92; 709/302, 300, 238, 245, 223

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,430,845	7/1995	Rimmer et al. .	
5,465,364	11/1995	Lathrop et al. .	
5,586,240	12/1996	Khan et al. .	
5,613,090 *	3/1997	Willems	709/302
5,615,401	3/1997	Harscoet et al. .	
5,680,618 *	10/1997	Freund	707/7
5,774,720 *	6/1998	Borgendale et al.	367/83
5,920,725 *	7/1999	Ma et al.	395/712

OTHER PUBLICATIONS

"Multiprotocol Management Agents: A Look At an Implementation and the Issues to Consider", Baktha Muralidharan, IEEE Journal on Selected Areas in Communications, vol. 11, No. 9, Dec. 1993.

"The Definition of Interoperability Architectures for Intelligent Devices Using Abstract Models", Elin L. Klaseen, et al., IEEE, (1995), pp. 237–245.

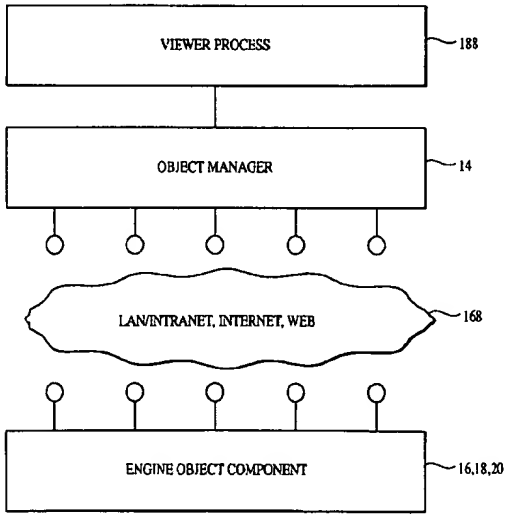
* cited by examiner

Primary Examiner—Joseph H. Feild
Assistant Examiner—Alford W. Kindred
(74) *Attorney, Agent, or Firm*—Irah H. Donner; Hale and Dorr LLP

(57) **ABSTRACT**

An image viewer process views at least one document image including an electronic document image, and performs viewing operations to the electronic document image. The process includes the step of selecting, by the user, one of a plurality of image viewing perspectives. Each of the plurality of image viewing perspectives provide the user the capability of viewing the document image in accordance with a different predefined user perspective. The process also includes the steps of selecting, by the user, using the image viewer process the document image to be viewed, and retrieving, by the image viewer process, the document image. The process also includes the step of displaying, by the image viewer process, the selected document image in accordance with an image viewing perspective selected by the user.

22 Claims, 23 Drawing Sheets



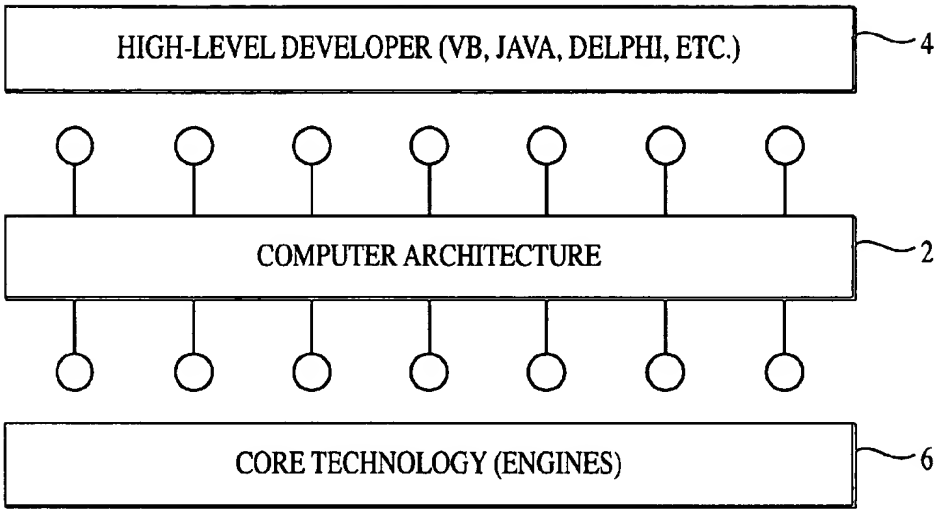

HIGH-LEVEL DEVELOPER (VB, JAVA, DELPHI, ETC.)
4
COMPUTER ARCHITECTURE
2
CORE TECHNOLOGY (ENGINES)
6

FIG. 1

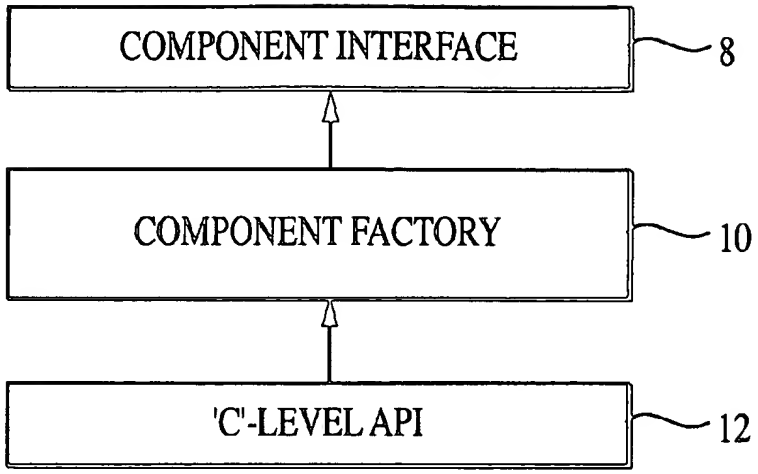

COMPONENT INTERFACE
8
COMPONENT FACTORY
10
'C'-LEVEL API
12

FIG. 2

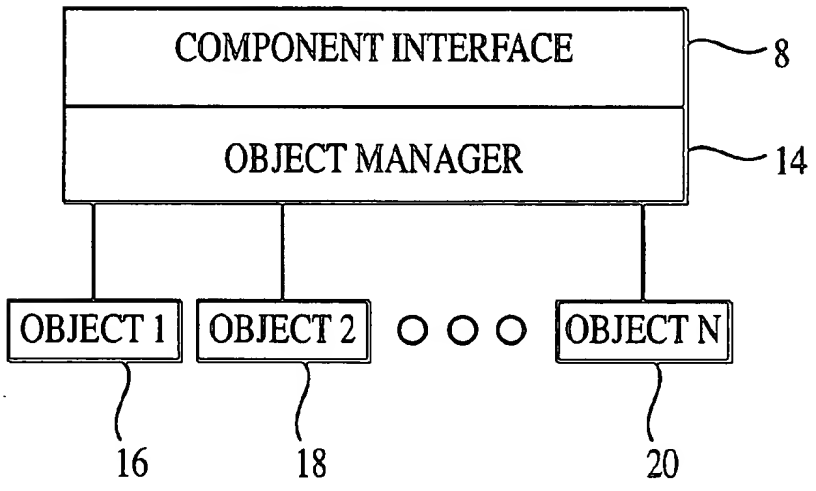

COMPONENT INTERFACE
8
OBJECT MANAGER
14
OBJECT 1
OBJECT 2
OBJECT N
16
18
20

FIG. 3

FIG. 4

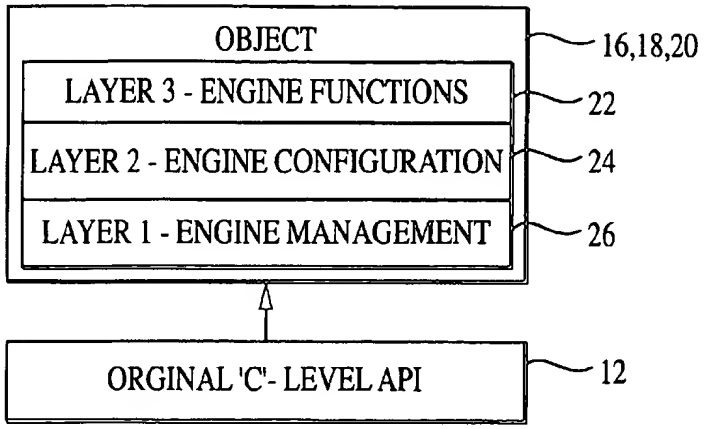


FIG. 5

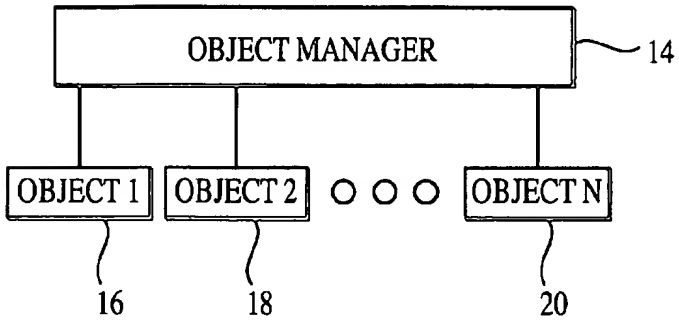
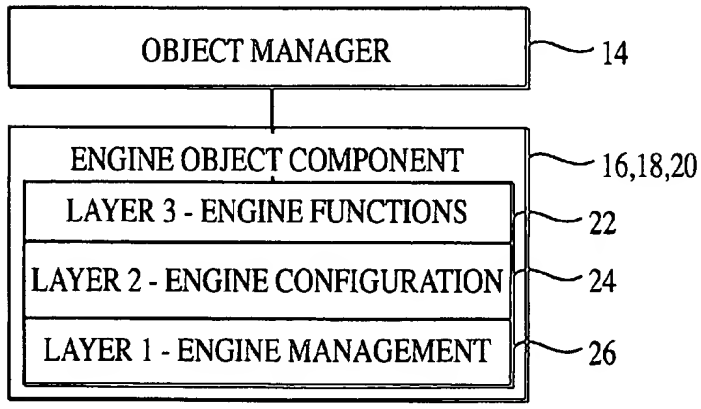


FIG. 6



IEngineManagement Interface	Arguments	Description
ActivatcEngine	BOOL Activate	Activates or deactivates an engine. This interface element will cause an engine to load itself to or unload itself from memory.
IsEngineActivated		Determine whether the engine has been successfully loaded into memory.

FIG. 7

IEngineManagement Interface	Arguments	Description
SetSetting	DWORD Setting VARIANT Value	Sets the setting Setting to a value of Value. the Setting argument is a unique number that represents a specific setting type. The Value argument is a union argument type that can accept any style argument, including an array of elements.
GetSetting	DWORD Setting VARIANT *Value	gets the setting Setting and places the value in Value. the Setting argument is a unique number that represents a specific setting type. The Value argument is a union argument type that can accept any style argument, including an array of elements.

FIG. 10

IEngineManagement Interface	Arguments	Description
Function	DWORD Setting VARIANT * Value	Initiate the function as represented by the Setting argument, using a variable number of arguments in using the Value array.

FIG. 12

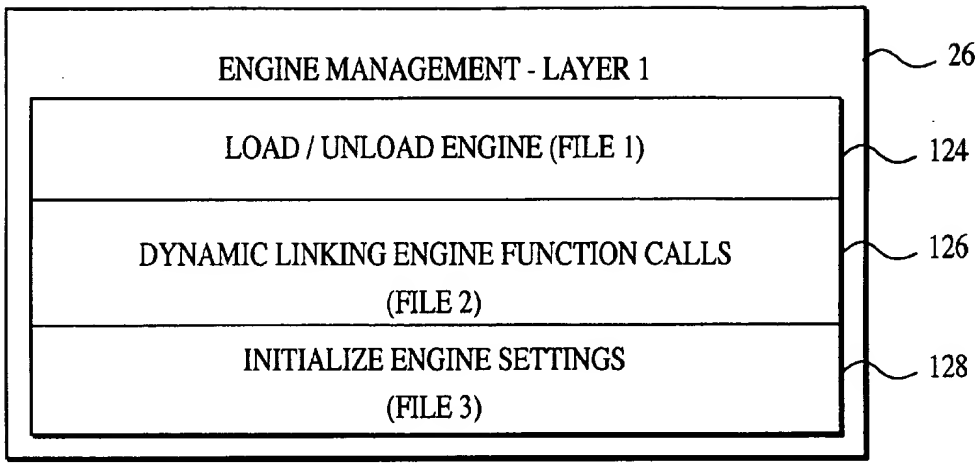

ENGINE MANAGEMENT - LAYER 1
26
LOAD / UNLOAD ENGINE (FILE 1)
124
DYNAMIC LINKING ENGINE FUNCTION CALLS
(FILE 2)
126
INITIALIZE ENGINE SETTINGS
(FILE 3)
128

FIG. 8

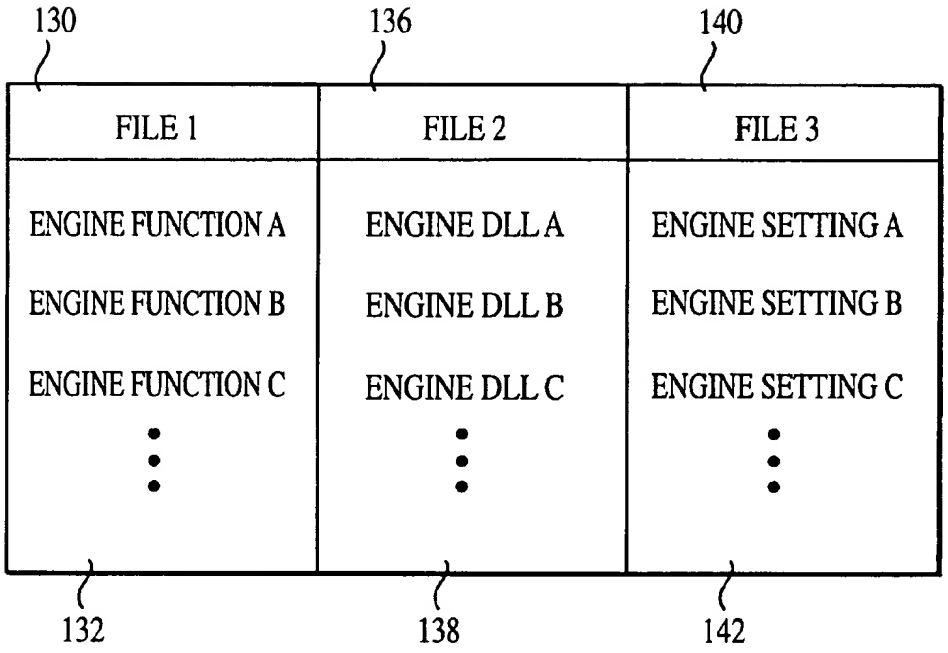

130
136
140
FILE 1
FILE 2
FILE 3
ENGINE FUNCTION A
ENGINE FUNCTION B
ENGINE FUNCTION C
ENGINE DLL A
ENGINE DLL B
ENGINE DLL C
ENGINE SETTING A
ENGINE SETTING B
ENGINE SETTING C
132
138
142

FIG. 9

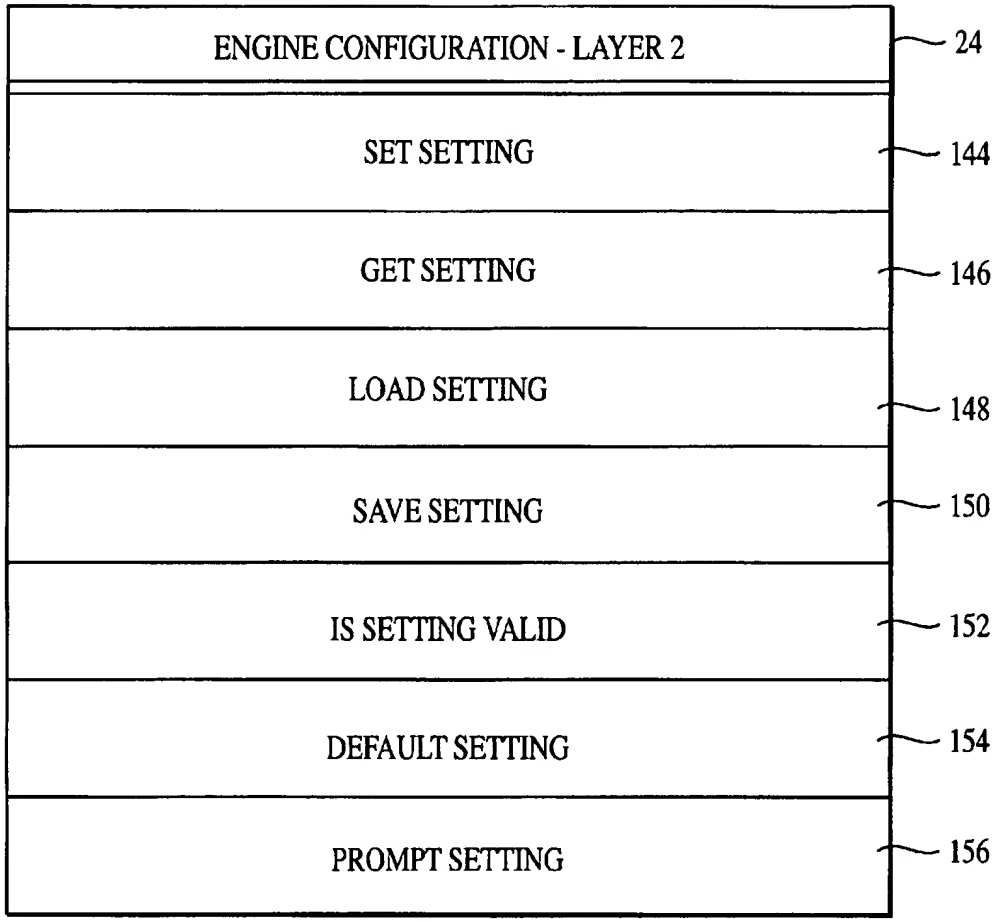

ENGINE CONFIGURATION - LAYER 2
24
SET SETTING
144
GET SETTING
146
LOAD SETTING
148
SAVE SETTING
150
IS SETTING VALID
152
DEFAULT SETTING
154
PROMPT SETTING
156

FIG. 11

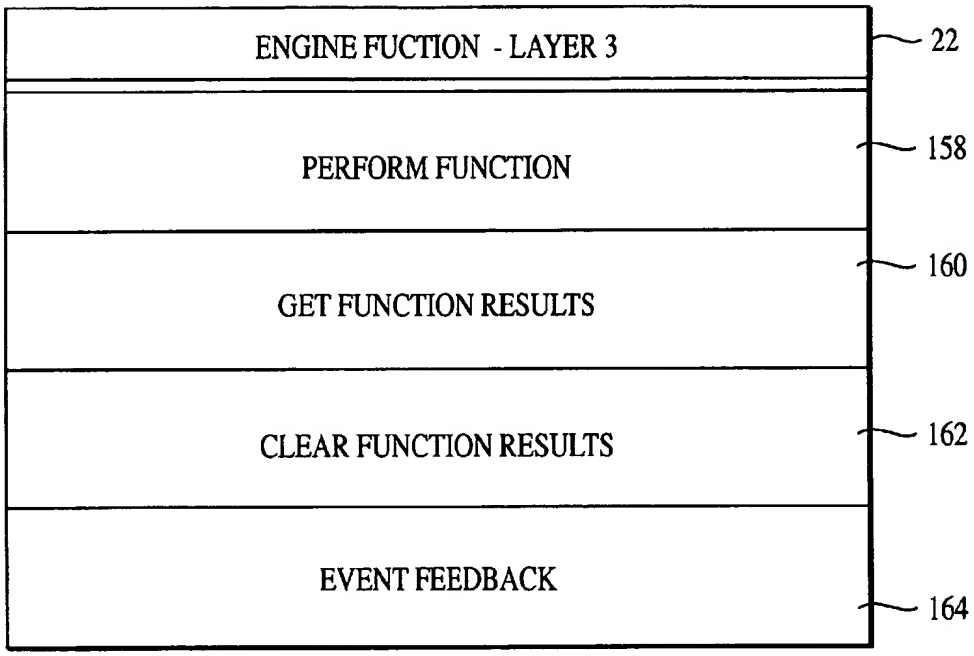

ENGINE FUCTION - LAYER 3
22
PERFORM FUNCTION
158
GET FUNCTION RESULTS
160
CLEAR FUNCTION RESULTS
162
EVENT FEEDBACK
164

FIG. 13

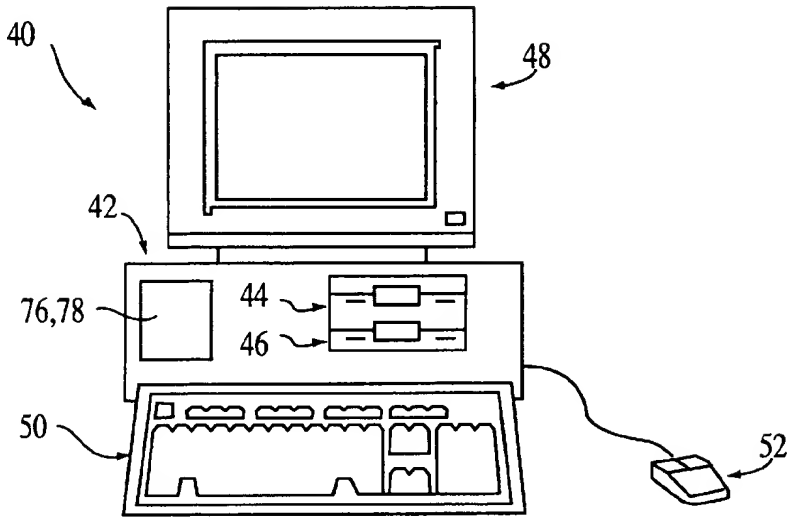

40
48
42
76,78
44
46
50
52

FIG. 14

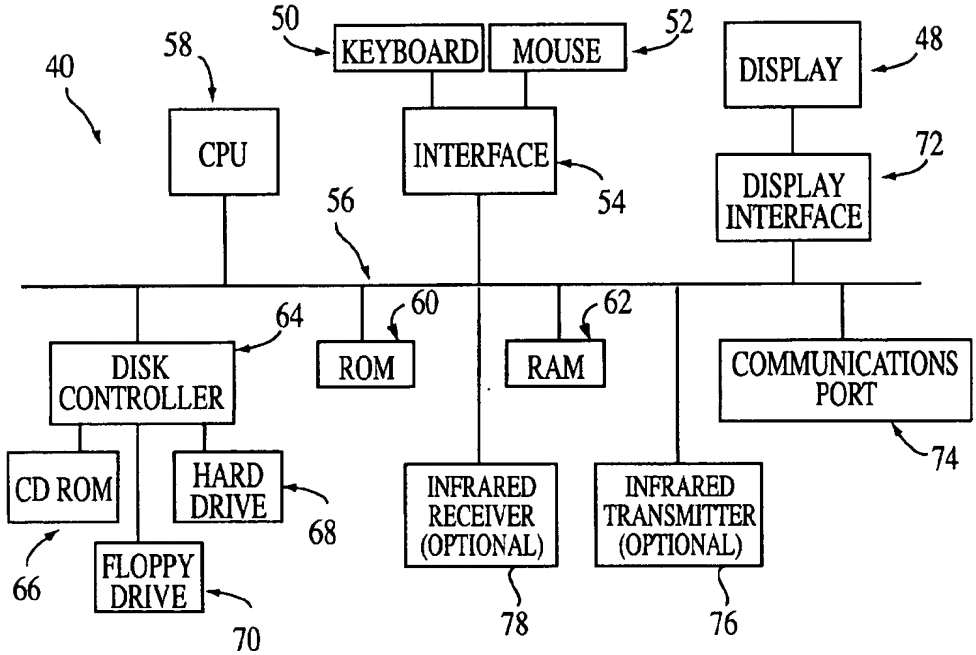

50
KEYBOARD
MOUSE
52
58
40
CPU
INTERFACE
54
56
DISPLAY
48
72
DISPLAY INTERFACE
64
60
62
DISK CONTROLLER
ROM
RAM
COMMUNICATIONS PORT
74
CD ROM
HARD DRIVE
68
INFRARED RECEIVER (OPTIONAL)
INFRARED TRANSMITTER (OPTIONAL)
66
FLOPPY DRIVE
70
78
76

FIG. 15

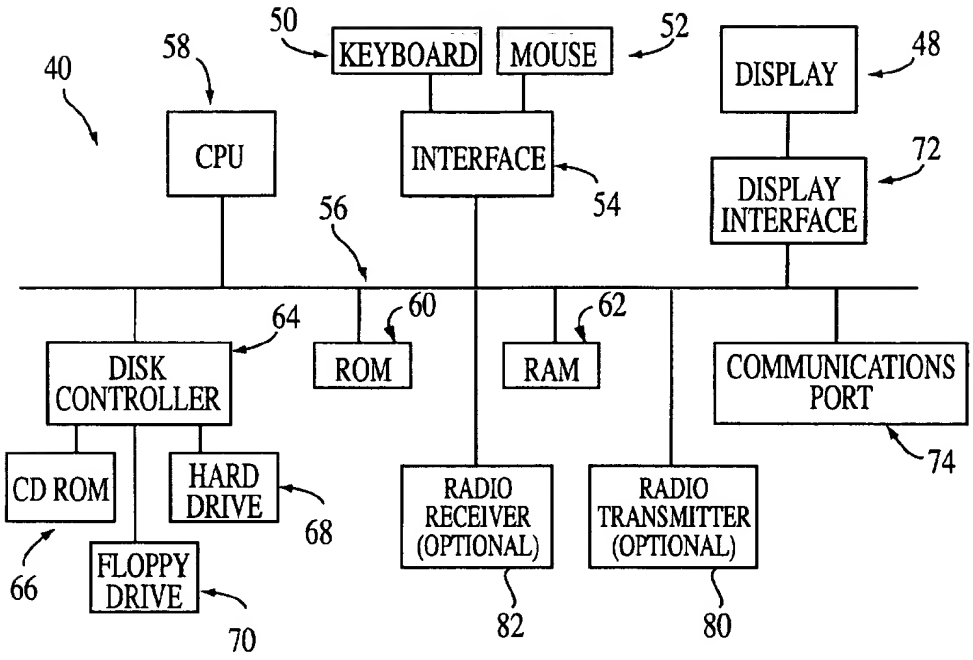

40
58
50
KEYBOARD
MOUSE
52
DISPLAY
48
CPU
INTERFACE
54
72
DISPLAY INTERFACE
56
64
60
62
DISK CONTROLLER
ROM
RAM
COMMUNICATIONS PORT
74
CD ROM
HARD DRIVE
68
RADIO RECEIVER (OPTIONAL)
RADIO TRANSMITTER (OPTIONAL)
66
FLOPPY DRIVE
70
82
80

FIG. 16

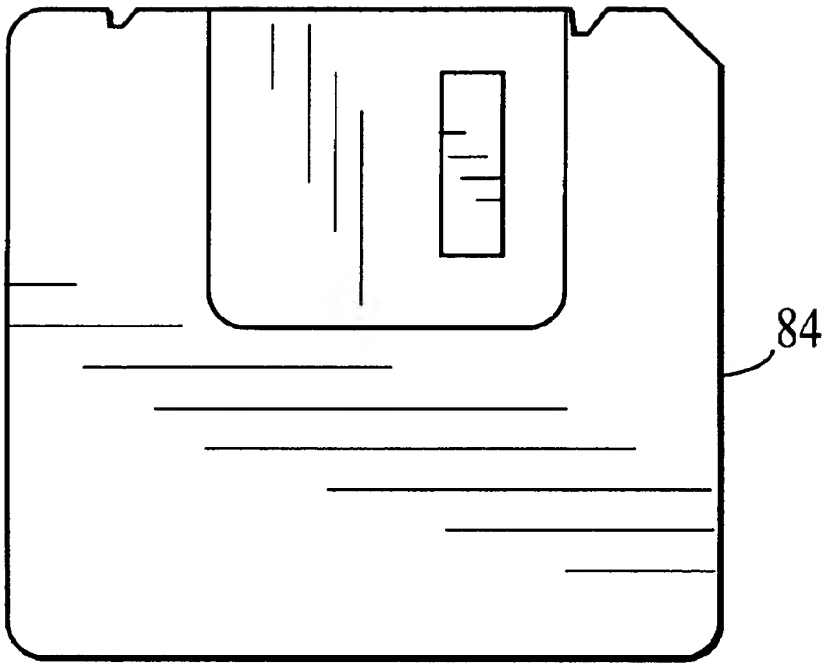


FIG. 17

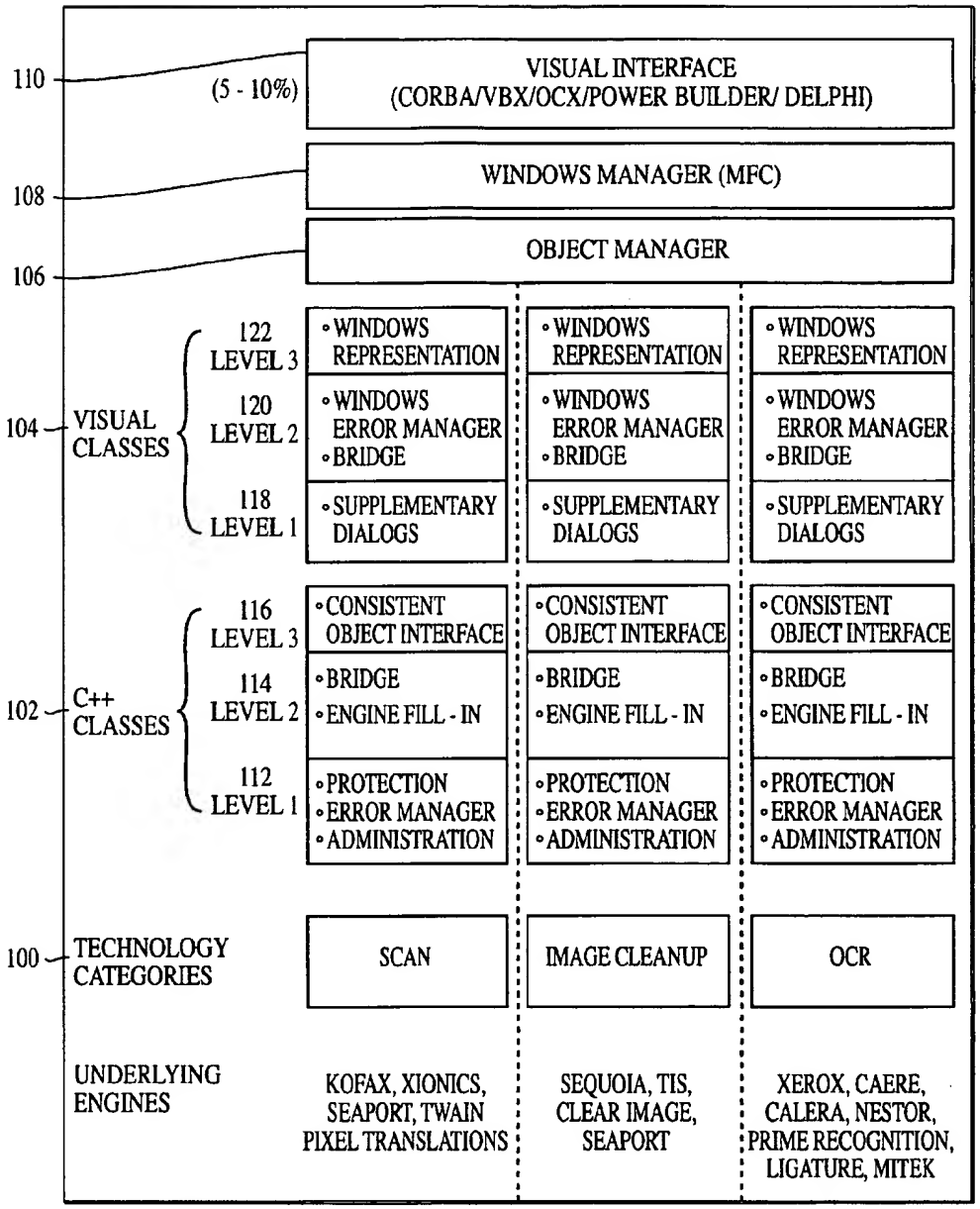

110
(5 - 10%)
VISUAL INTERFACE
(CORBA/VBX/OCX/POWER BUILDER/ DELPHI)
108
WINDOWS MANAGER (MFC)
106
OBJECT MANAGER
104
VISUAL CLASSES
122 LEVEL 3
120 LEVEL 2
118 LEVEL 1
• WINDOWS REPRESENTATION
• WINDOWS ERROR MANAGER
• BRIDGE
• SUPPLEMENTARY DIALOGS
102
C++ CLASSES
116 LEVEL 3
114 LEVEL 2
112 LEVEL 1
• CONSISTENT OBJECT INTERFACE
• BRIDGE
• ENGINE FILL - IN
• PROTECTION
• ERROR MANAGER
• ADMINISTRATION
100
TECHNOLOGY CATEGORIES
SCAN
IMAGE CLEANUP
OCR
UNDERLYING ENGINES
KOFAX, XIONICS, SEAPORT, TWAIN PIXEL TRANSLATIONS
SEQUOIA, TIS, CLEAR IMAGE, SEAPORT
XEROX, CAERE, CALERA, NESTOR, PRIME RECOGNITION, LIGATURE, MITEK

FIG. 18

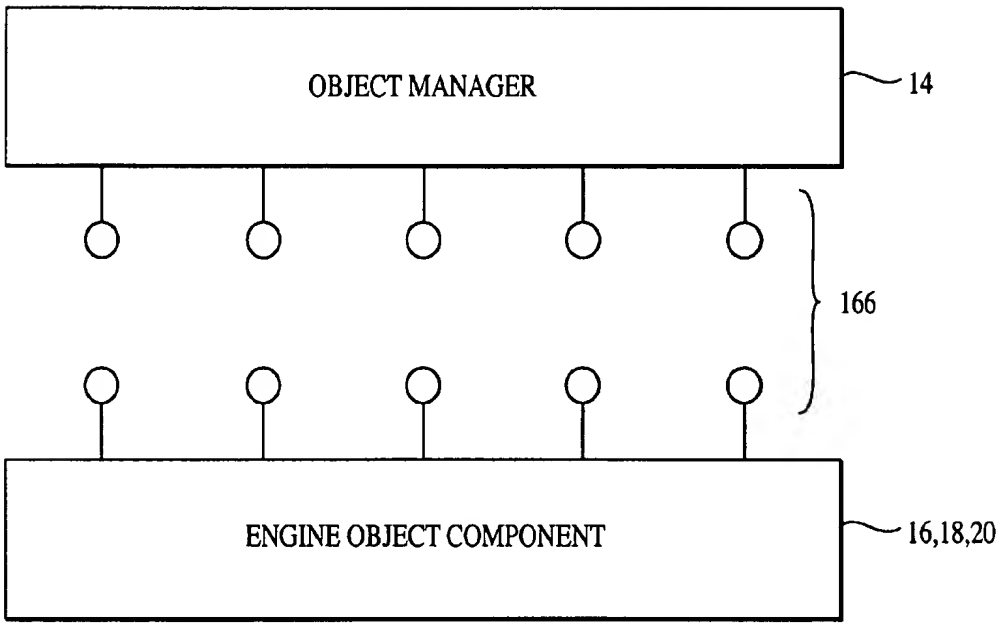


FIG. 19

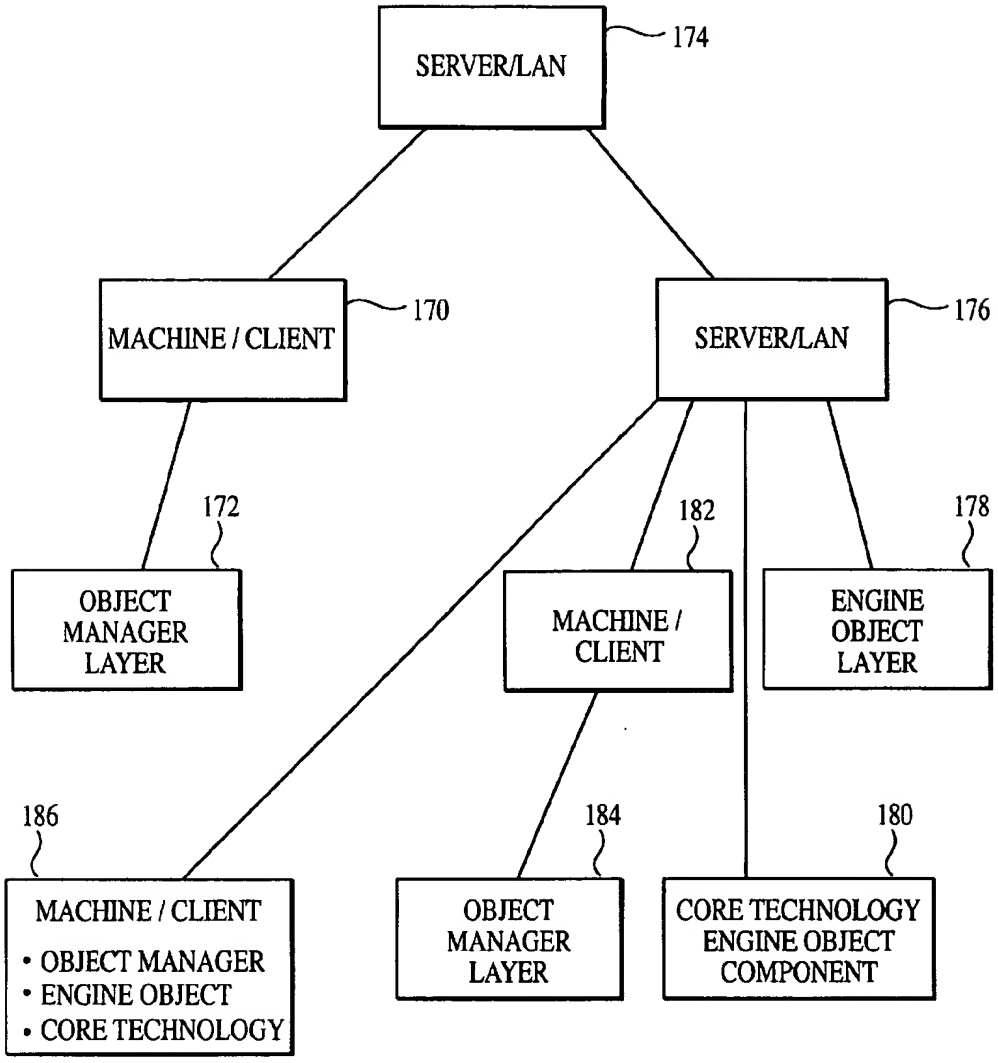

SERVER/LAN
174
MACHINE / CLIENT
170
SERVER/LAN
176
172
OBJECT MANAGER LAYER
182
MACHINE / CLIENT
178
ENGINE OBJECT LAYER
186
MACHINE / CLIENT
• OBJECT MANAGER
• ENGINE OBJECT
• CORE TECHNOLOGY
184
OBJECT MANAGER LAYER
180
CORE TECHNOLOGY ENGINE OBJECT COMPONENT

FIG. 20

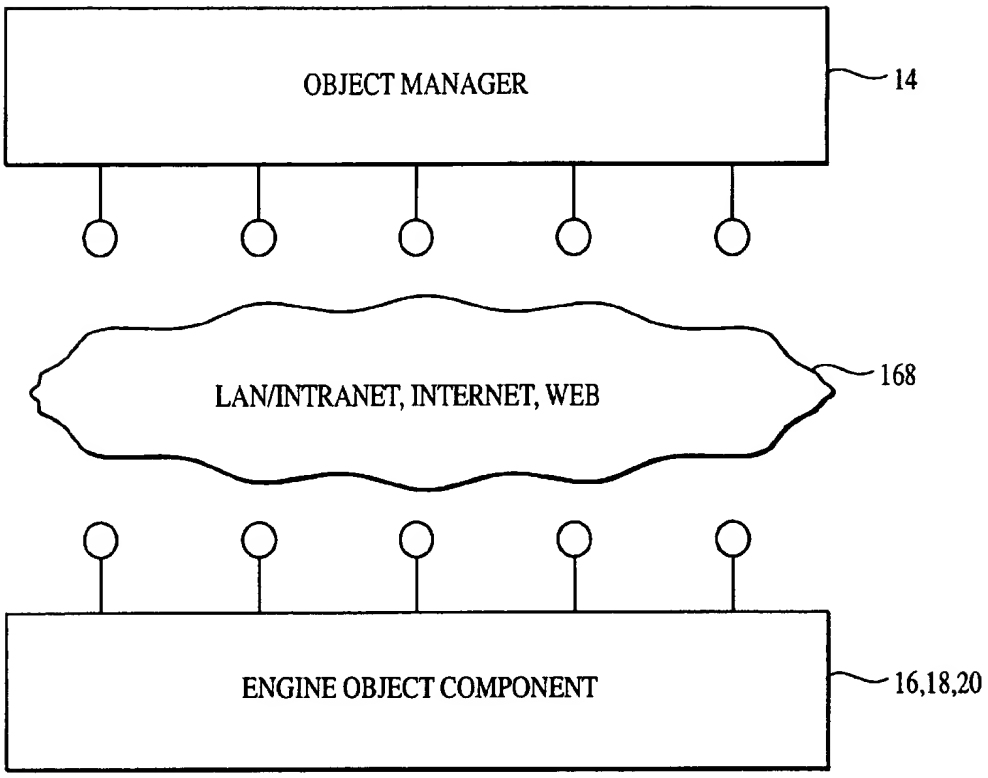

OBJECT MANAGER
14
LAN/INTRANET, INTERNET, WEB
168
ENGINE OBJECT COMPONENT
16,18,20

FIG. 21

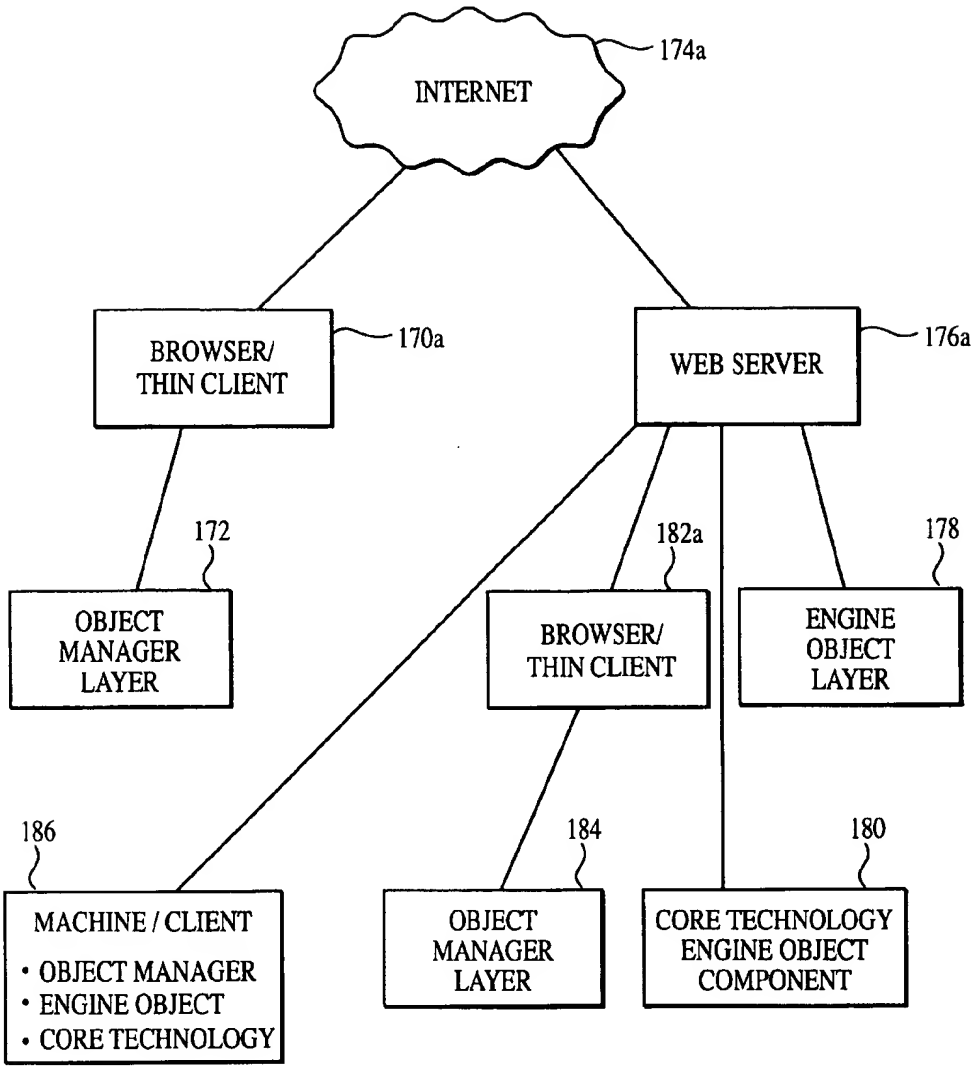

INTERNET
174a
BROWSER/
THIN CLIENT
170a
WEB SERVER
176a
172
OBJECT
MANAGER
LAYER
182a
BROWSER/
THIN CLIENT
178
ENGINE
OBJECT
LAYER
186
MACHINE / CLIENT
• OBJECT MANAGER
• ENGINE OBJECT
• CORE TECHNOLOGY
184
OBJECT
MANAGER
LAYER
180
CORE TECHNOLOGY
ENGINE OBJECT
COMPONENT

FIG. 22

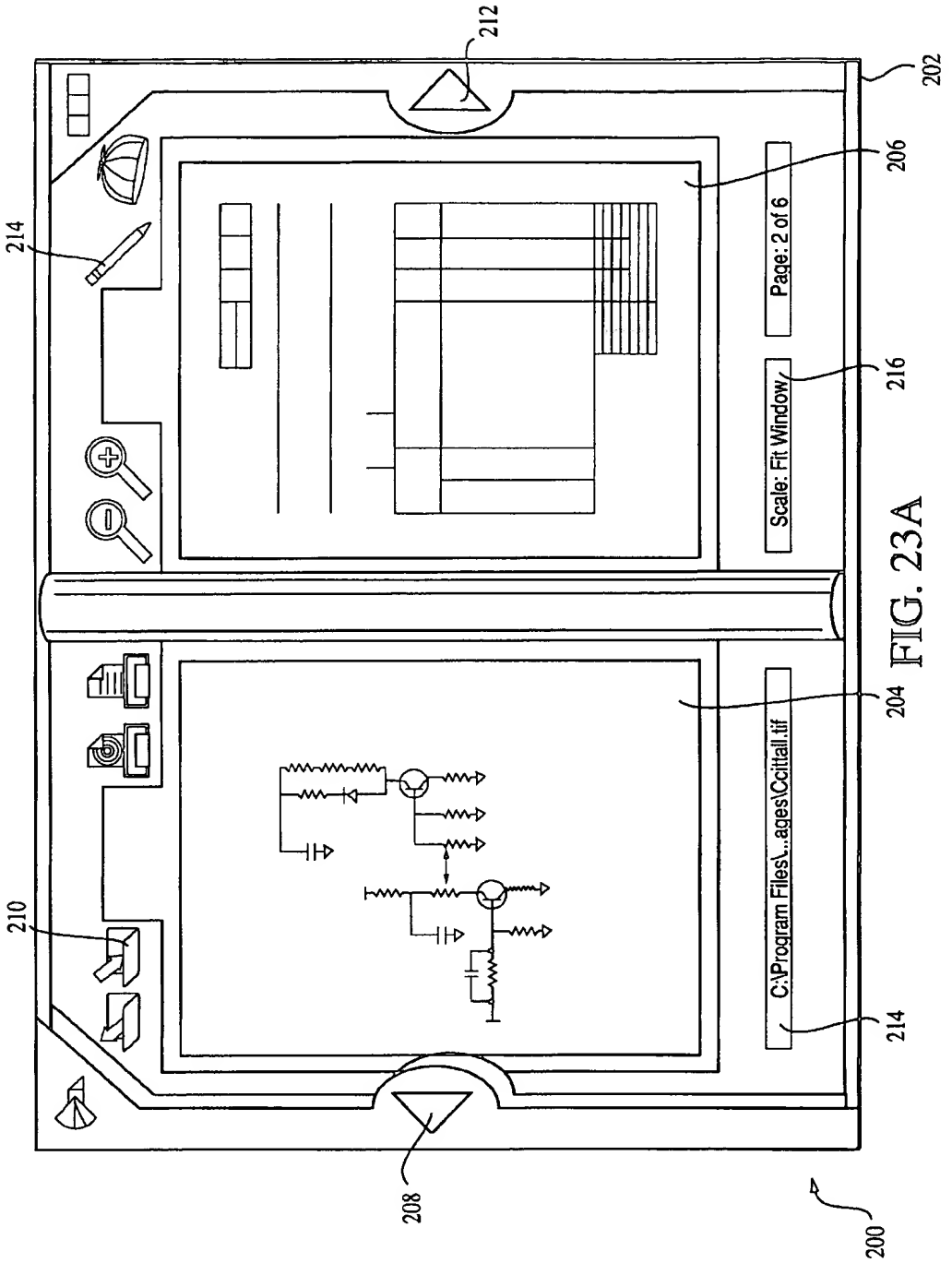

212
202
206
214
Page: 2 of 6
216
Scale: Fit Window
204
C:\Program Files\...ages\Ccittall.tif
214
210
208
200

FIG. 23A

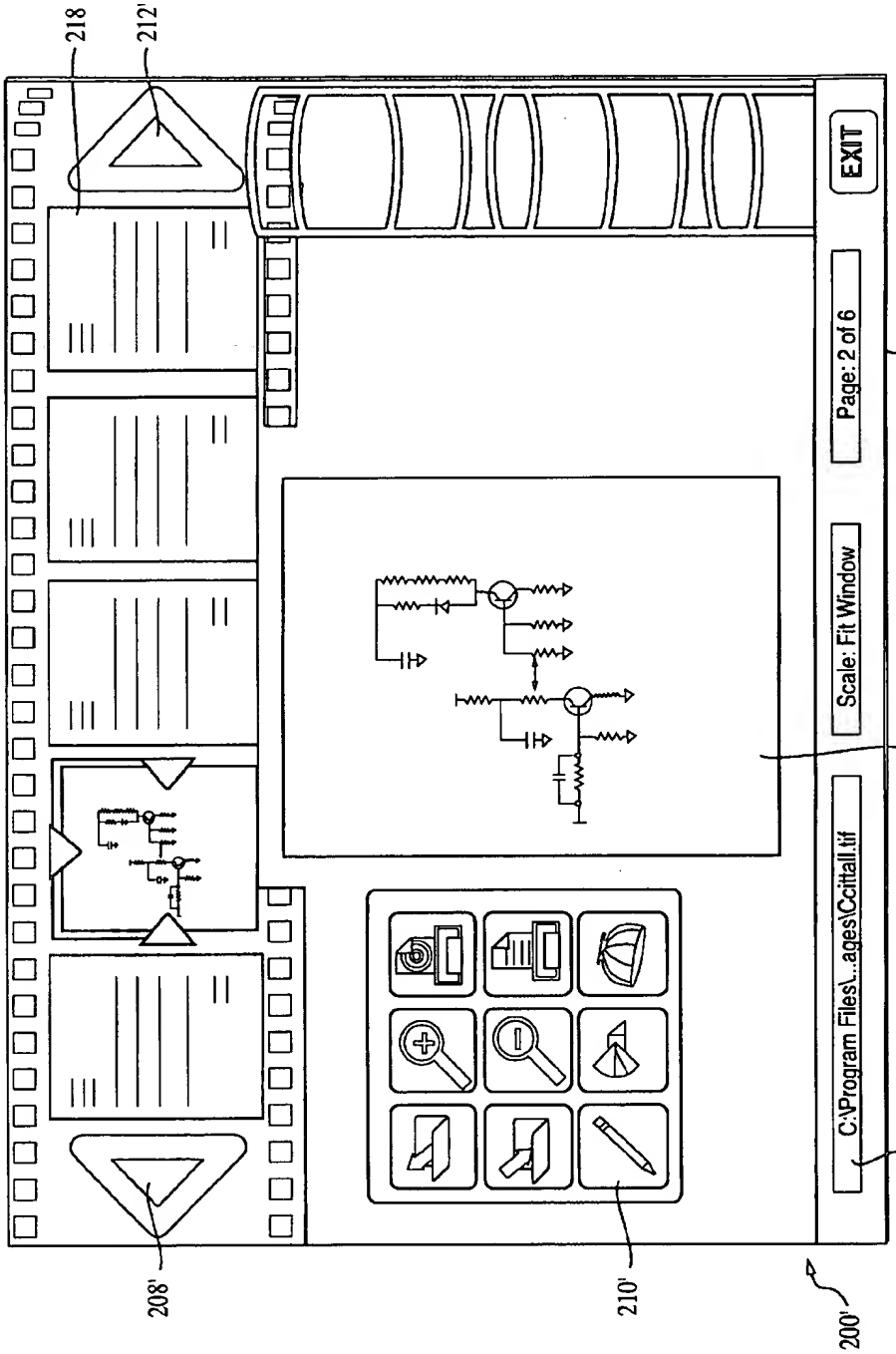

218
212'
EXIT
Page: 2 of 6
202'
Scale: Fit Window
204'
C:\Program Files\...ages\Ccittall.tif
214'
208'
210'
200'

FIG. 23B

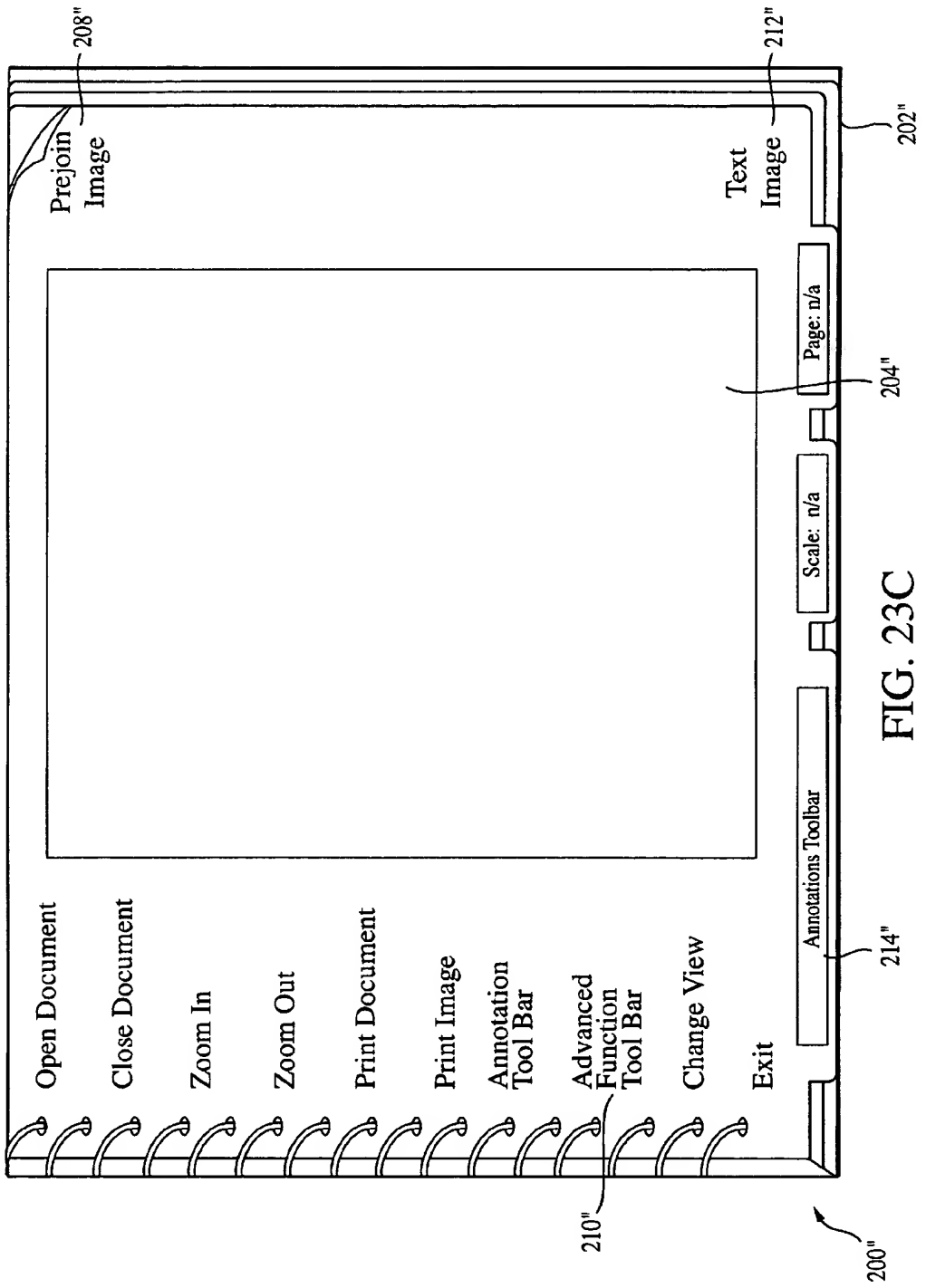

208"
Prejoin Image
212"
Text Image
202"
Page: n/a
204"
Scale: n/a
Annotations Toolbar
214"
Open Document
Close Document
Zoom In
Zoom Out
Print Document
Print Image
Annotation Tool Bar
Advanced Function Tool Bar
Change View
Exit
210"
200"

FIG. 23C

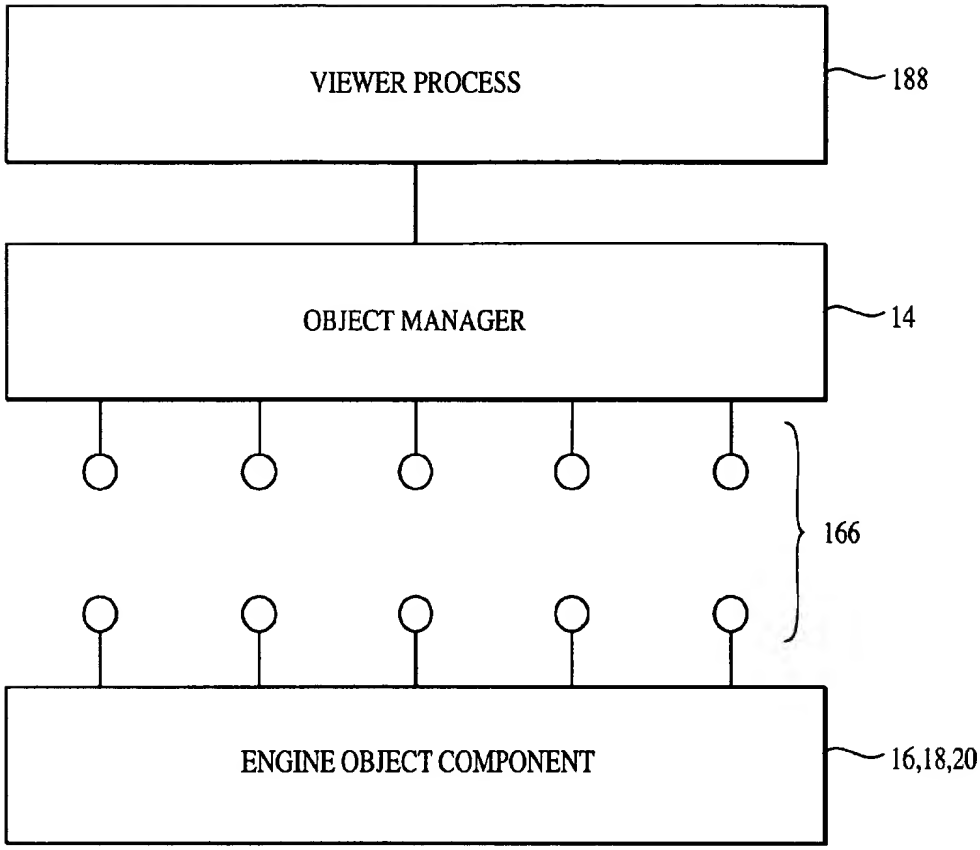

VIEWER PROCESS
188
OBJECT MANAGER
14
166
ENGINE OBJECT COMPONENT
16,18,20

FIG. 24

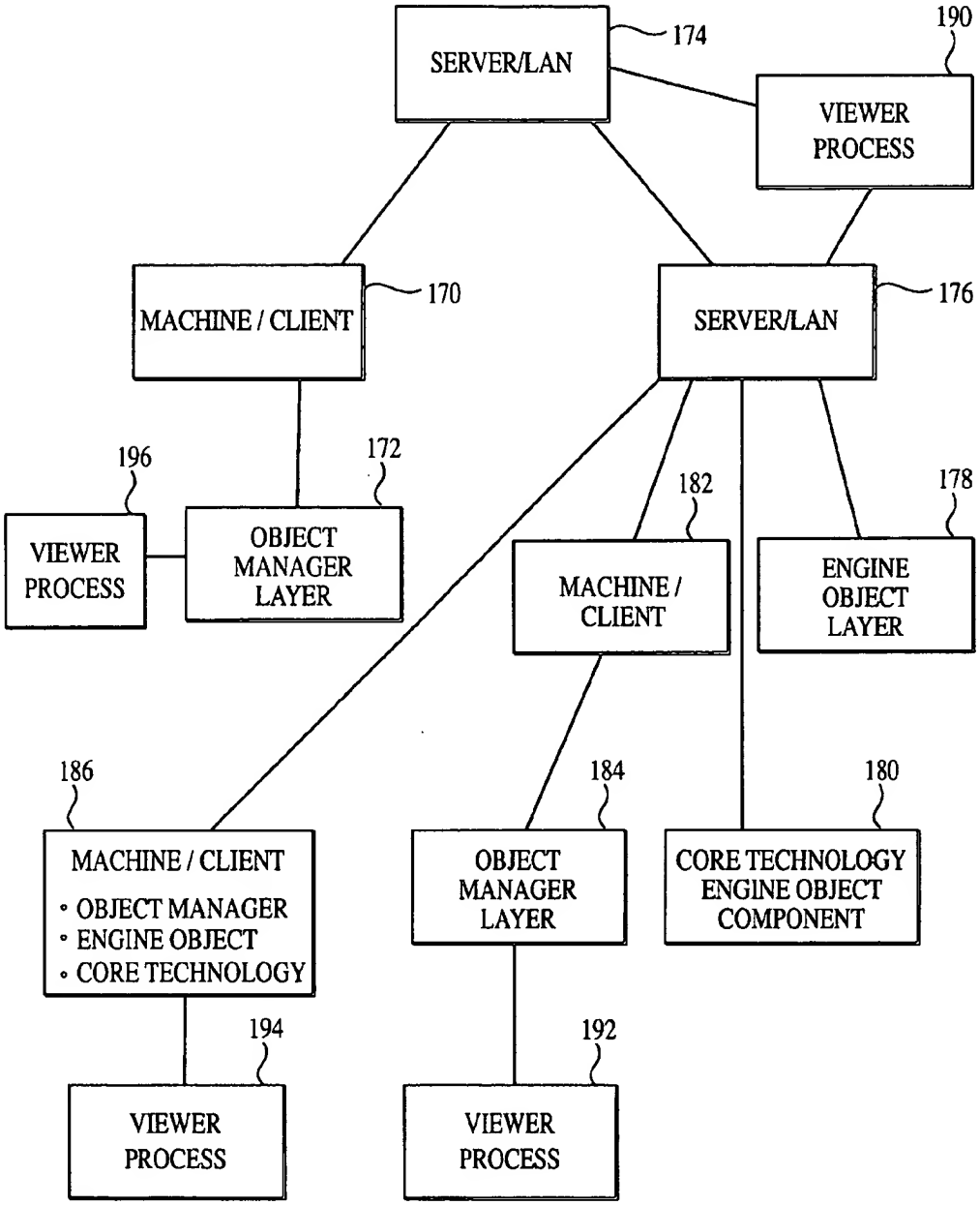

SERVER/LAN
174
190
VIEWER PROCESS
MACHINE / CLIENT
170
SERVER/LAN
176
196
172
182
178
VIEWER PROCESS
OBJECT MANAGER LAYER
MACHINE / CLIENT
ENGINE OBJECT LAYER
186
184
180
MACHINE / CLIENT
• OBJECT MANAGER
• ENGINE OBJECT
• CORE TECHNOLOGY
OBJECT MANAGER LAYER
CORE TECHNOLOGY ENGINE OBJECT COMPONENT
194
192
VIEWER PROCESS
VIEWER PROCESS

FIG. 25

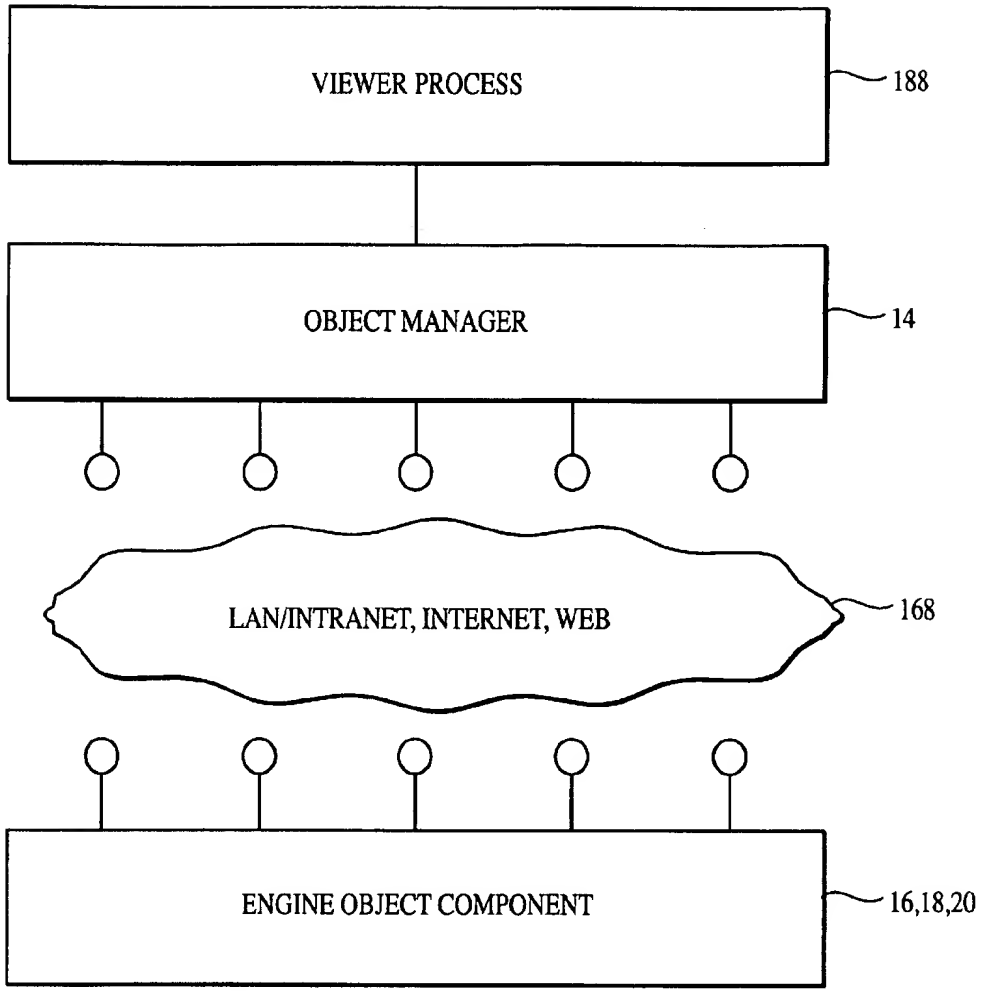

VIEWER PROCESS
188
OBJECT MANAGER
14
LAN/INTRANET, INTERNET, WEB
168
ENGINE OBJECT COMPONENT
16,18,20

FIG. 26

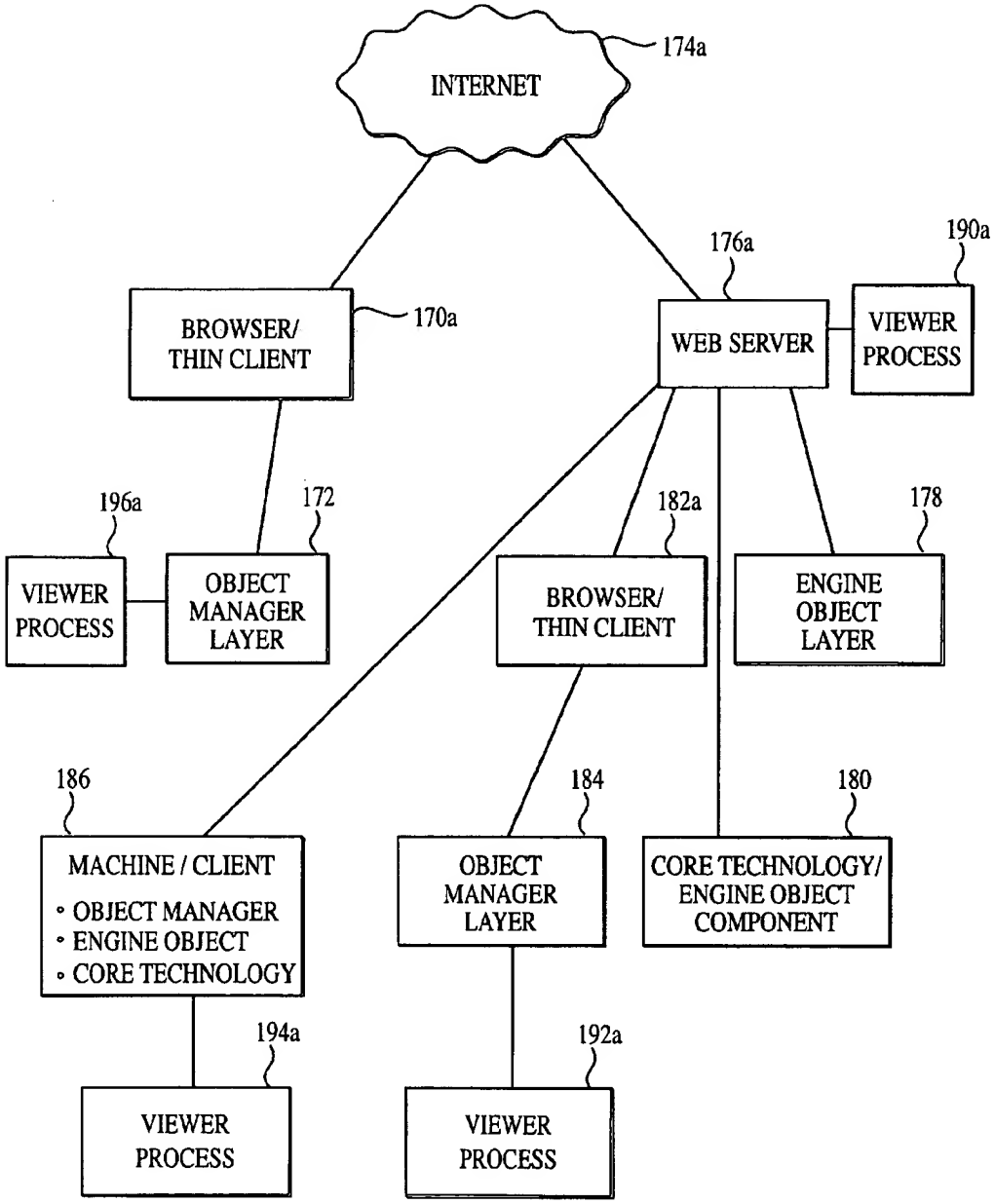

INTERNET
174a
BROWSER/
THIN CLIENT
170a
176a
WEB SERVER
190a
VIEWER
PROCESS
196a
VIEWER
PROCESS
172
OBJECT
MANAGER
LAYER
182a
BROWSER/
THIN CLIENT
178
ENGINE
OBJECT
LAYER
186
MACHINE / CLIENT
• OBJECT MANAGER
• ENGINE OBJECT
• CORE TECHNOLOGY
184
OBJECT
MANAGER
LAYER
180
CORE TECHNOLOGY/
ENGINE OBJECT
COMPONENT
194a
VIEWER
PROCESS
192a
VIEWER
PROCESS

FIG. 27

PROCESS AND ARCHITECTURE FOR USE ON STAND-ALONE MACHINE AND IN DISTRIBUTED COMPUTER ARCHITECTURE FOR CLIENT SERVER AND/OR INTRANET AND/OR INTERNET OPERATING ENVIRONMENTS

RELATED APPLICATIONS

This application is a continuation-in-part application of U.S. patent application Ser. No. 08/911,083 filed on Aug. 14, 1997, to the same applicant, and hereby incorporated by reference.

This application is related to, and claims priority from, the following copending provisional applications, the details of which are hereby incorporated by reference: COMPUTER PROCESS FOR IMAGE CONTROL, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial Number 60/028,129; COMPUTER PROCESS FOR IMAGE CONTROL, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial Number 60/028,522; COMPUTER PROCESS FOR ENGINE BENCHMARKING, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial Number 15 60/028,128; COMPUTER PROCESS FOR DESK TOP IMAGING, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial Number 60/028, 697; COMPUTER PROCESS FOR IMAGE VIEWING, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial Number 60/028,639; COMPUTER PROCESS FOR IMAGE CAPTURE, Applicant: Laurence C. Klein, Filed: Oct. 18, 1996, Serial No. 60/028,685.

FIELD OF THE INVENTION

The present invention is generally related to a computer architecture and process for image viewing in a stand-alone and/or distributed environment, and more particularly to a computer architecture and process for image viewing using a substantially uniform management in a stand-alone and/or distributed computing environment including, for example, client server and/or intranet and/or internet operating environments.

BACKGROUND OF THE RELATED ART

A "C" or "C++"-Level API (hereinafter "C" Level), which is the native language and interface for a vast repository of core technologies from small software vendors and research laboratories, are unique to each designer. The designer of a text retrieval "C"-API will generally implement an interface that is completely different than a second inventor creating a "C"-level API for OCR.

Every "C"-level API is unique, both in its choice of API syntax as well as its method for implementing the syntax. Some API's consist of one or two functions that take parameters offering options for different features offered by the technology. Other APIs consist of hundreds of functions with few arguments, where each function is associated with a particular feature of the core technology. Other APIs provide a mixture of some features being combined with one function with many arguments, while other features are separated into individual function calls.

Without any constraints, each designer of a core technology chooses to implement his or her technology with an interface that is suitable to the subject or simply was the most expedient choice of the moment. Since there are no constraints, a "C"-level API has a totally unpredictable interface that can often be the hindrance to using the core technology.

Additionally, every API manages errors differently further complicating the problems described above. Some APIs return a consistent error code for each function. Error management in this case is very organized and manageable. Other APIs return error codes as one of the parameters passed to the function. There are APIs that mix the choice of error management and have some functions return an error code while other functions pass the error code as a parameter of a function. Errors can also be managed by a callback function, eliminating the need for passing any error code as part of the function. In some instances of a poorly implemented API the errors are not passed back at all.

Every engine, such as a text retrieval or an OCR (Optical Character Recognition) engine, has a unique interface. This interface is generally a "C"-level API (Application Program Interface). Further, an API can at any time be synchronous, asynchronous, manage one or more callbacks, require input, pass back output, carry a variety of different styles of functions, return values or not return values, and implement the unpredictable. This unpredictability in APIs further compounds the problem of developing a sane way of interfacing between components and APIs.

To date, because of the complexities of "C"-level APIs and components interfacing thereto, the only way to create a component out of an existing "C"-level API is to have an experienced programmer in the field to do the work. Humans can intelligently analyze an API, and create a component based on intelligent decisions and experiences. In most cases, the learning curve for understanding and integrating a new engine can be one man-month to several man-years and generally requires highly experienced "C" programmers. Requiring a human to perform the necessary work is costly, and subject to real-life human constraints.

Since there is no structure or format for implementing "C"-level APIs, the ability to automatically transform a unique API into a standard component would seem impossible, since that would take a nearly-human level of intelligence.

I have determined that a component factory, if it is to be truly automated or manually expedited, must be able to take any "C"-level API and transform it into a component.

I have also determined an efficient and workable design for an architecture to define the migration path for any "C"-level API into a component.

I have also determined that it is desirable to develop software tools for automatically generating reusable software components from core software technologies, thus making these software technologies available to a much larger user base.

I have further determined that it is desirable to design a distributed computer architecture and process for manually and/or automatically generating reusable software components. The computer architecture may be implemented using a client server and/or intranet and/or internet operating environments.

I have further determined that it is desirable to design a computer architecture and process for image viewing in a stand-alone and/or distributed environment. The computer architecture and process optionally uses a substantially uniform management layer in a stand-alone and/or distributed computing environment including, for example, client server and/or intranet and/or internet operating environments.

SUMMARY OF THE INVENTION

One would expect the translating a "C"-level API from its native state into a component would require human-level

intelligence. This is mainly because "C"-level APIs have virtually no constraints as to how they can be implemented. This means that there are an infinity variations of APIs, which can only be managed by human-level intelligence. While this point is true, I have determined that the appropriate solution starts at the other side of the equation, which is the component itself.

My solution starts out with a definition of a component that can sustain the feature/function requirements of any API. In other words, the interface of a generic component can be defined such that the features and functions of virtually any API can be re-implemented within its bounds. The two known end-points are, for example, the "C"-level API that generally starts with each component (although other programming languages may also be used and are within the scope of the present invention), and the component interface that represents any set of features/functions on the other side. The component factory migrates the original "C"-level API from its original state into the generic interface defined by the topmost layer. The first feature that can be demonstrated is that there is a topmost layer that can define a component interface that can represent the features/functions of most core technologies.

The component factory migrates the "C"-level API to the topmost level. Doing this in one large step would be impossible since the "C"-level API has a near-infinite variety of styles. However, the architecture advantageously has enough well-defined and well-structured layers for implementing the topmost component interface, for creating the component factory.

The computer architecture is designed for managing a diverse set of independent core technologies ("engines") using a single consistent framework. The architecture balances two seemingly opposing requirements: the need to provide a single consistent interface to many different engines with the ability to access the unique features of each engine.

The benefit of the architecture is that it enables a company to rapidly "wrap" a sophisticated technology so that other high-level developers can easily learn and implement the core technology. The computer architecture is therefore a middleware or enabling technology.

Another benefit of the architecture is that it provides a high-level specification for a consistent interface to any core technology. Once a high-level developer learns the interface described herein for one engine, that knowledge is easily transferable to other engines that are implemented using the architecture. For example, once a high-level developer learns to use the computer architecture for OCR (Optical Character Recognition), using the computer architecture for other engines, such as barcode recognition or forms processing, is trivial.

The architecture described herein is, at once, a framework for rapidly wrapping sophisticated technologies into high-level components, as well as a framework for high-level developers to communicate with a diverse set of engines. The creating of a component factory is based on the fact that the architecture defines a clear path for "wrapping" any C-level API into a component using simple structures and many rote steps. This process is currently being done in an inefficient manner by a programmer in the field.

In addition, the method described herein for creating a component factory creates a well-defined multi-tiered architecture for a component and automates, substantially automates, or manually expedites hereinafter automate the process of migrating a "C"-API from its native state through the various tiers of the architecture resulting in a standardized component. Advantageously, the method described herein does not base the component factory on making human-level intelligent decisions on how to translate a "C"-API into a component. Rather, by creating a well-defined architecture described below that is multi-tiered, the method is a series of incremental steps that need to be taken to migrate the "C"-API from one tier within the architecture to the next. In this way each incremental step is not a major one, but in sequence the entire series of steps will result in a component.

Since each step of migration is not a major one, the chances for automating these steps is significantly higher and the likelihood of being able to create the component factory becomes feasible. This approach is in fact what makes the method cost-effective, since the alternative approach, i.e., computer-generated human-level decision making, has many years before becoming sophisticated enough to replace humans in any realistic decision-making process.

The main features of the architecture are twofold:

1) Defining system architecture that describes in detail how to implement a component from a "C"-level API;
2) Creating a component factory by automating the migration of a "C"-level API from one tier within the architecture to the next. The latter feature is the key to actually making the component factory feasible. With a fixed architecture that can be used to implement a "C"-level API as a component (using a programmer), that same architecture can be used as the basis for the component factory model.

In order to make the component factory, each step of the architecture needs to be designed to facilitate automation or manually expedited. In other words, I have determined that automating/expediting the process of taking the original "C"-level API and migrating it to a Level 1 layer, and then a Level 1 to a Level 2, and then a Level 2 to a Level 3 layer, and so on, the component has been implemented automatically or more efficiently. The component factory is therefore a sum of the ability to automate migrating the "C"-level API from one layer to the next within a well-defined architecture for implementing components.

There are numerous core technologies, such as text-retrieval and ICR (Intelligent Character Recognition), that have already been implemented, and are only available as "C"-level APIs. Many, if not most, core technologies are first released exclusively as "C"-level APIs. While there are integrators and corporations who have the team of technologists who can integrate these "C"-level APIs in-house, most companies are looking for component versions that can be implemented at a much higher level.

Therefore, many of the core technologies that are only available in a "C"-level API are not being used due to their inaccessible interface. The benefit of the component factory is that it can rapidly make available core technologies implemented as "C" APIs that would otherwise be underutilized or dormant in research labs by converting them to high-level components that can be used by millions of power-PC users.

With the advent of the World Wide Web (WEB) this opportunity has increased exponentially. The WEB is now home to a vast number of WEB authors with minimal formal training who can implement HTML pages and build web sites. One of the fundamental technologies for extending the capability of the WEB from simple page viewing to interactive and sophisticated applications is components.

A component extends the capability of HTML by enabling a WEB author to add core technology as a pre-packaged

technology. Since components are fundamental to the growth and usability of the WEB, having a component factory that can translate "C"-level toolkits into components that are then usable within WEB sites opens a vast and new worldwide market to these technologies.

It is a feature and advantage of the present invention to implement a component factory, that is automated or manually expedited.

It is another feature and advantage of the present invention to be able to take any "C"-level API and transform it into a component.

It is another feature and advantage of the present invention to define an efficient and workable design for an architecture to provide the migration path for any C-level API into a component.

It is another feature and advantage of the present invention to develop software tools for automatically generating reusable software components from core software technologies.

It is another feature and advantage of the present invention to develop software tools to make software components available to a much larger user base.

It is another feature and advantage of the present invention in providing a distributed computer architecture and process for manually and/or automatically generating reusable software components.

It is another feature and advantage of the present invention in providing a distributed computer architecture and process for manually and/or automatically generating reusable software components where the computer architecture is implemented using a client server and/or intranet and/or internet operating environments.

It is another feature and advantage of the present invention in providing a computer architecture and process for image viewing in a stand-alone and/or distributed environment.

It is another feature and advantage of the present invention in providing a computer architecture and process that uses a substantially uniform management layer in a stand-alone and/or distributed computing environment including, for example, client server and/or intranet and/or internet operating environments.

The present invention is based, in part, on my discovery that it is possible to make the component factory, and that each step of the architecture is designed to facilitate automation or manually design of components. The present invention is also based, in part, on my discovery that by automating/expediting the process of taking the original "C"-level API and migrating it to a Level 1 layer, and then a Level 1 to a Level 2, and then a Level 2 to a Level 3 layer, and so on, the component has been implemented automatically and/or more manually efficiently. The component factory is therefore a sum of the ability to automate migrating the "C"-level API from one layer to the next within a well-defined architecture for implementing components.

The present invention is also based, in part, on my discovery that the object manager and engine object component layers may be advantageously be designed to operate independently, thereby making possible a distributed computing environment, as described below in detail. I have further discovered that an efficient method of implementing the engine object component layer is by using pre-populated tables/files. I have further discovered that the engine management layer may be advantageously divided into a three layer structure of load/unload engine, dynamic linking engine function calls, and initialize engine setting.

In accordance with one embodiment of the invention, a computer implemented process migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an object for each engine. The object provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented process includes the step of providing an engine management function interfacing with the program specific API. The engine management function furnishes a protective wrapper for each function call associated with the engine, trapping errors, and provides error management and administration to prevent conditions associated with improper engine functioning. The process optionally includes the step of providing an engine configuration function transforming API calls received from the program specific API into standardized calls. The engine configuration function provides additional functionality, including safely loading and unloading the engine. The process optionally includes the step of providing an engine function managing the standardized calls for each engine, thereby providing substantially uniform access to the engine and the engine settings associated with the engine.

In accordance with another embodiment of the invention, a computer implemented method migrates at least one program specific Application Programmer Interface (API) from an original state into a generic interface by building an object for each engine. The object provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented method includes the steps of defining a substantially consistent interface for individual object components that represent diverse technologies, and migrating a plurality of engines to the consistent interface. The computer implemented method also includes the step of substantially automatically and/or substantially uniformly, managing the individual object components using a predefined object manager and the consistent interface.

In accordance with another embodiment of the invention, a computer architecture migrates at least one program specific Application Programmer Interface (API) from an original state into a generic interface by building an object for each engine. The object provides substantially uniform access to the engine and engine settings associated with the engine. The computer architecture includes an engine management layer interfacing with the program specific API and providing engine management and administration, an engine configuration layer transforming API calls received from the program specific API into standardized calls, and an engine layer managing the standardized calls for each engine.

In accordance with another embodiment of the invention, an engine management layer configures a computer architecture to perform one or more computer implemented or computer assisted operations. The computer operations include one or more of loading and unloading engine dynamic link libraries into and out of memory for each engine, mapping at least one engine function to at least one corresponding engine object, providing general error detection and error correction for each engine, determining and matching arguments and returning values for mapping the at least one engine function to the at least one corresponding engine object, and/or managing error feedback from the at least one program specific API.

In accordance with another embodiment of the invention, a distributed computer system migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an object for each engine. The object provides substantially uniform access to the engine and engine settings associated with the engine. The distributed computer system includes a server config-

ured to include at least one engine having an engine interface providing one or more features to be executed, and at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine. The distributed computer system also includes at least one client configured to be connectable to the server and optionally configured to be connectable to another server. The client includes an object manager layer communicable with and managing the at least one engine component stored on the server via the substantially consistent interface.

In accordance with another embodiment of the invention, a distributed computer implemented process migrates a program specific Application Programmer Interface (API) from an original state into a generic interface by building an object for each engine. The object provides substantially uniform access to the engine and engine settings associated with the engine. The computer implemented process includes the step of providing, on a server, at least one engine having an engine interface, and providing one or more features to be executed. The computer implemented process also includes the step of providing, on at least one of the server and another server connectable to the server, at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine. The computer implemented process also includes the step of providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an object manager layer communicable with and managing the at least one engine component via the substantially consistent interface.

In accordance with another embodiment of the invention, an image viewer process views at least one document image including an electronic document image, and performs viewing operations to the electronic document image. The process includes the step of selecting, by the user, one of a plurality of image viewing perspectives. Each of the plurality of image viewing perspectives provide the user the capability of viewing the document image in accordance with a different predefined user perspective. The process also includes the steps of selecting, by the user, using the image viewer process the document image to be viewed, and retrieving, by the image viewer process, the document image. The process also includes the step of displaying, by the image viewer process, the selected document image in accordance with an image viewing perspective selected by the user.

In accordance with another embodiment of the invention, a computer readable tangible medium is provided that stores the process thereon, for execution by the computer.

In accordance with another embodiment of the invention, a computer readable tangible medium is provided that stores an object thereon, for execution by the computer.

These together with other objects and advantages which will be subsequently apparent, reside in the details of construction and operation as more fully herein described and claimed, with reference being had to the accompanying drawings forming a part hereof wherein like numerals refer to like elements throughout.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is an illustration of the placement and/or use of the computer architecture and/or method of the present invention;

FIG. **2** is an illustration of the component factory migrating the original "C"-level API from its original state into the generic interface defined by the topmost layer;

FIG. **3** is an overview of the computer architecture in the present invention;

FIG. **4** is an illustration of the design of an Object in accordance with the computer architecture of the present invention;

FIG. **5** is an illustration of the architecture comprised of two major parts;

FIG. **6** is an illustration of the architecture of an engine component including, for example, three layers designed to migrate the original API of the engine to a consistent COM interface;

FIG. **7** is a table illustrating the engine management specification with definitions;

FIG. **8** is an illustration of the engine management layer being divided into three functions/specifications;

FIG. **9** is an illustration of exemplary tables used to drive the three functions of the engine management layer illustrated in FIG. **8**;

FIG. **10** is an exemplary table illustrating the engine configuration specification with definitions;

FIG. **11** is another exemplary table illustrating the engine configuration specification;

FIG. **12** is an exemplary table illustrating the engine functionality specification with definitions;

FIG. **13** is another exemplary table illustrating the engine functionality specification;

FIG. **14** is an illustration of a main central processing unit for implementing the computer processing in accordance with a computer implemented embodiment of the present invention;

FIG. **15** illustrates a block diagram of the internal hardware of the computer of FIG. **14**;

FIG. **16** is a block diagram of the internal hardware of the computer of FIG. **15** in accordance with a second embodiment;

FIG. **17** is an illustration of an exemplary memory medium which can be used with disk drives illustrated in FIGS. **14–16**;

FIG. **18** is an illustration of another embodiment of the component factory migrating the original "C"-level API from its original state into the generic interface defined by the topmost layer;

FIG. **19** is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for client server and/or intranet operating environments;

FIG. **20** is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for client server and/or intranet operating environments;

FIG. **21** is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for network environments, such as the Internet;

FIG. **22** is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components in the Internet environment;

FIGS. **23A–23C** are illustrations of the image viewer user interface and/or functionality associated therewith in accordance with the present invention;

FIG. **24** is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments;

FIG. **25** is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments;

FIG. **26** is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in network environments, such as the Internet; and

FIG. **27** is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in the Internet environment.

NOTATION AND NOMENCLATURE

The detailed descriptions which follow may be presented in terms of program procedures executed on a computer or network of computers. These procedural descriptions and representations are the means used by those skilled in the art to most effectively convey the substance of their work to others skilled in the art.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be noted, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in most cases, in any of the operations described herein which form part of the present invention; the operations are machine operations. Useful machines for performing the operation of the present invention include general purpose digital computers or similar devices.

The present invention also relates to apparatus for performing these operations. This apparatus may be specially constructed for the required purpose or it may comprise a general purpose computer as selectively activated or reconfigured by a computer program stored in the computer. The procedures presented herein are not inherently related to a particular computer or other apparatus. Various general purpose machines may be used with programs written in accordance with the teachings herein, or it may prove more convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description given.

BEST MODE FOR CARRYING OUT THE INVENTION

The purpose of the computer architecture and process described herein is to create a component factory that can automatically generate reusable software components from sophisticated core software technologies. Many, if not most, core software technologies, such as OCR (Optical Character Recognition) or barcode recognition, are designed and implemented using a "C"-language API (Application Program Interface). The technology is often complex, requiring months of trial-and-error to correctly develop application systems using the technology. While there are millions of Intranet developers and power-PC users who are capable of assembling component-based systems, I have determined that there are relatively few "C" programmers (estimated at less than 100,000) who can learn and implement application software with these complex 'C'-level API's. It is therefore desirable to develop software tools for automatically generating reusable software components from core software technologies thus making these software technologies available to a much larger user base.

Since I have determined that there is no structure or format for implementing "C"-level API's, the ability to automatically transform a unique API into a standard component would seem impossible since that would take a nearly-human level of intelligence. To date, the only way, I am aware, to create a component out of an existing API is to have an existing programmer in the field do the work for each API. Humans can intelligently analyze an API and create a component based on intelligent decisions tempered by experience. The challenge of creating a component factory is the challenge of partially or substantially recreating the component design and formulating effective implementation decisions.

One would expect the translating a "C"-level API from its native state into a component would require human-level intelligence. This is mainly because "C"-level APIs have virtually no constraints as to how they can be implemented. This means that there are an infinity variations of APIs, which can only be managed by human-level intelligence. While this point is true, I have determined that the appropriate solution starts at the other side of the equation, which is the component itself.

My solution starts out with a definition of a component that can sustain the feature/function requirements of any API. In other words, the interface of a generic component can be defined such that the features and functions of virtually any API can be re-implemented within its bounds. The two known end-points are the "C"-level API that started with, and the component interface that represents any set of features/functions on the other side.

I have also determined that one solution for creating a computer architecture and process for implementing a component factory is to create a well-defined multi-tiered systems architecture for a component and to automate, substantially automate, or manually expedite from its native state through the various tiers of the systems architecture resulting in a standardized or substantially standardized component. Advantageously, this solution is not based on making human-level intelligent decisions on how to translate a 'C'-level API into a component. Rather, by starting with a well-defined systems architecture that is multi-tiered, a series of incremental steps that migrates a C-level API from one tier within the systems architecture to the next may be performed, and which are facilitated using the architecture and/or process described herein.

Advantageously, each incremental step is not a major one, but in sequence the entire series of steps will result in a usable component. Since each step of migration is not a major one, the chances of automating these steps is significantly higher and the likelihood of being able to create the component factory becomes more feasible.

The fundamental building blocks of the computer architecture and process are twofold:

1) To define a systems architecture that describes in detail how to implement a component from a C-level API
2) To create a component factory by automating, substantially automating, or manually expediting the migration of a C-level API from one tier within the architecture to the next.

The building blocks are the keys or important to actually making the component factory feasible.

Significantly, the computer architecture and processes described herein have application to the Intranet and document market marketplace. Corporations are embracing internet computing technologies to create enterprise-level Intranets and Extranets. Using standard browser technologies, corporations and government entities are rapidly adopting the internet computing model and are developing enterprise applications by assembling standard Microsoft specified Active X components. These are not "C" programmers; rather they are typical power PC users. Further availability of reusable components would only fuel this development.

The general outline for creating a component factory is described below in detail. It is important to note that automatically, substantially automatically, or manually building a component is neither obvious nor guaranteed. As will be described below in detail, automating or substantially automating the building of a component consists of automating individual steps that comprise the component architecture. However, in today's application environment, any amount of automation will dramatically increase the efficiencies of building a component

The computer architecture is designed for managing a diverse set of independent core technologies ("engines") using a single consistent framework. The architecture balances two seemingly opposing requirements: the need to provide a single consistent interface to many different engines with the ability to access the unique features of each engine.

The benefit of the architecture is that it enables a company to rapidly "wrap" a sophisticated technology so that other high-level developers can easily learn and implement the core technology. The computer architecture is therefore a middleware or enabling technology, as illustrated in FIG. **1**.

As illustrated in FIG. **1**, computer architecture **2**, described below in detail, is a middle layer between high level developer programs **4** (such as C-level APIs, or other programs having similar characteristics) and are technology/component engines **6** (such as OCR, bar code recognition, and other components having similar characteristics).

Another benefit of the architecture is that it provides a high-level specification for a consistent interface to any core technology. Once a high-level developer learns the interface described herein for one engine, that knowledge is easily transferable to other engines that are implemented using the architecture. For example, once a high-level developer learns to use the computer architecture for OCR (Optical Character Recognition), using the computer architecture for other engines, such as barcode recognition or forms processing, is trivial.

In summary, the architecture and process described herein is at once a framework for rapidly wrapping sophisticated technologies into high-level components, as well as a framework for high-level developers to communicate with a diverse set of engines. The creating of a component factory is based on the fact that the architecture defines a clear path for "wrapping" any C-level API into a component using simple structures and many rote steps. This process is currently being done in an inefficient manner by a programmer in the field.

The method described herein for creating a component factory creates a well-defined multi-tiered architecture for a component and automates, substantially automates, or manually expedites (hereinafter "automates") the process of migrating a "C"-API from its native state through the various tiers of the architecture resulting in a standardized component.

Advantageously, the method described herein does not base the component factory on making human-level intelligent decisions on how to translate a "C"-level API into a component. Rather, by creating a well-defined architecture described below that is multi-tiered, the method is a series of incremental steps that need to be taken to migrate the "C"-level API from one tier within the architecture to the next. In this way each incremental step is not a major one, but in sequence the entire series of steps will result in a component.

Since each step of migration is not a major one, the chances for automating these steps is significantly higher and the likelihood of being able to create the component factory becomes feasible. This approach is in fact what makes the method cost-effective, since the alternative approach, i.e., computer-generated human-level decision making, is currently unavailable and would require much effort, if at all possible, to replace humans in any realistic decision-making process.

With a fixed architecture that can be used to implement a "C"-level API as a component (using a programmer), that same architecture can be used as the basis for the component factory model. In order to make the component factory, each step of the architecture needs to be designed to facilitate automation or manually expedited. In other words, I have determined that automating/expediting the process of taking the original "C"-level API and migrating it to a Level 1 layer, and then a Level 1 to a Level 2, and then a Level 2 to a Level 3 layer, and so on, the component has been implemented automatically, or more efficiently manually. The component factory is therefore a sum of the ability to automate migrating the "C"-level API from one layer to the next within a well-defined architecture for implementing components.

As illustrated in FIG. **2**, the component factory **10** migrates the original "C"-level API **12** from its original state into the generic interface **8** defined by the topmost layer. The first feature that can be demonstrated is that there is a topmost layer **8** that can define a component interface that can represent the features/functions of most core technologies. The component factory **10** migrates the "C"-level API **12** to the topmost level **8**. Doing this in one large step would be impossible since the "C"-level API has a near-infinite variety of styles. However, the architecture advantageously has enough well-defined and well-structured layers for implementing the topmost component interface, for creating the component factory.

A simplified overview of the architecture is illustrated in FIG. **3**. In FIG. **3**, the component interface **8** sits on top of an Object Manager **14** that communicates with individual objects e.g., **16, 18, 20**. These objects **16, 18, 20** represent specific core technologies that are represented as "C"-level APIs. The design of Object**1**, Object**2**, . . . ObjectN is illustrated in FIG. **4**.

A component factory can be created by automating the process of migrating the original "C"-level API **12** from its original state to the Layer 1—Engine Management tier **26**, and then from the state to Layer 2 Engine configuration tier **24**, and so on up the Engine Functions layer **22**. These layers will be further described below.

The computer architecture is implemented, for example, as a standard COM component, as an ActiveX control; the specifications designed by Microsoft, published in the technical literature, and incorporated herein by reference. ActiveX control (COM) support is currently available within any Microsoft 32-bit Windows operating environment. ActiveX controls are supported by all OLE-based

applications, including all of Microsoft's end-user products (e.g., Microsoft Office, Word, Access, Powerpoint, Access), the main Internet Browsers (Microsoft's Internet Explorer and Netscape's Navigator—the latter with an add-in product and by 4Q97 directly), most other name-brand end-user Windows products (e.g., Lotus Notes), and all major development environments (e.g., Microsoft Visual Basic and Visual C++, Delphi, Borland C++, Power Builder). By implementing the architecture as, for example, an ActiveX control, complex technologies can be programmed by virtually any Windows or Intranet user or developer. Of course, other component specifications may also be used.

Although the architecture has been implemented as a COM-based technology with C++ as the language of choice, the architecture can be implemented in many other languages (e.g. Java) and distributed architectures (e.g. CORBA).

Every engine, such as a text retrieval or an OCR (Optical Character Recognition) engine, has a unique interface. This interface is generally a "C"-level API (Application Program Interface). In most cases, the learning curve for understanding and integrating a new engine can be a one man-month to several man-years and generally requires highly experienced "C" programmers. The purpose of the architecture is to define a clear infrastructure within which any core can be rapidly "wrapped" so that users and developers can have easy access to these core technologies.

In addition to defining the infrastructure for engines to be accessible to typical users, the architecture also defines how to migrate an engine from its native state to the prescribed interface. In other words, the architecture goes beyond simply defining the framework for wrapping engines, it also defines the specific steps for wrapping these engines.

The architecture consists of a hierarchical series of layers that take any "C"-level API from its unique state to one that is standard and consistent. The result is a single, highly-integrated object component that contains and manages any type of engine that can be programmed regardless of the nature and subject of the core technology. The architecture therefore not only defines the goal (e.g., the object component interface) but also the means of implementing that goal for any type of engine.

The architecture is comprised of two major parts as illustrated in FIG. **5**: the Object Manager **14**, and the individual object components **16, 18, 20**. The Object Manager **14** in FIG. **5** manages individual object components **16, 18, 20** illustrated as Object **1**, Object **2**, etc. The Object Manager **14** communicates with the individual object components **16, 18, 20** using a consistent COM interface.

Each object component implements the feature set of an individual engine by mapping a consistent COM interface to the "C"-Level API interface of the individual engine that it supports. In this way the Object Manager can consistently communicate with each engine, using the engine's object component. Because the COM interface of each object component is consistent, the Object Manager can interface with every underlying engine the same way.

The features of the architecture include:

1) definition of consistent COM interfaces for individual object components that represent diverse technologies;
2) a prescribed process for migrating any engine to the defined consistent COM interface; and/or
3) a predefined Object Manager that automatically manages the individual object components.

When implemented, for example, as an ActiveX control, the architecture also yields an umbrella control that can be used by a high-level programmer to program and manage numerous sophisticated technologies in a plug-and-play environment. In order to facilitate the discussion of the architecture itself it is best to start with the architecture of the engine object component and then describe the Object Manager. Since the Object Manager is directly dependent on the engine object components, an understanding of the latter will assist in the description of the former.

Engine Object Component—**16, 18, 20**

The purpose of the engine object is to wrap a specific engine using a series of layers that convert the engine's unique interface into a COM interface that is, for example, specified by the architecture. The architecture not only defines the consistent COM interface for implementing an engine, it also describes how to implement the interface from the original "C"-Level API. Once the COM interface of the engine object component is implemented, the Object Manager understands and can therefore communicate with it.

Each engine component consists of, for example, three layers that are designed to migrate the original API of the engine to a consistent COM interface. As illustrated in FIG. **6**, the Object Manager **14** communicates with the topmost layer **22** of the object component **16, 18, 20** which is the defined interface of object component.

Each layer is described below in two parts. The first part is the prescribed COM interface for communicating with the engine object component. The second part describes a specific path for automating building the layer. By providing an outline for automating building each layer, the overall engine object component can be automatically, substantially automatically or manually expedited and generated.

Layer 1—Engine Management **26**

The first layer in the object component architecture is designed to deal with the fundamental features of an engine. This includes the ability to load and unload the standard or commercially available via, for example, MicroSoft Corporation, engine Dynamic Link Libraries (DLLs) into memory, as well as the ability to consistently deal with errors. This is the most fundamental layer because it is the essential "wrapper" layer of an engine. Once this layer is complete all interaction with the underlying engine is filtered through this layer. Additional important engine management functions include dynamically accessing a function call of an engine, and initializing engine settings. All of these engine management functions are optionally and beneficially table driven to promote or facilitate access to, and implementation of, engine management functions.

The Layer **1** specification is summarized in FIG. **7** that describes the IEngineManagement COM interface. The purpose of the IEngineManagement interface is to transparently load and unload an engine to and from memory. I have determined that this is often the core feature that is incorrectly implemented and a cause for hard-to-find bugs. This layer may be generated manually by a developer who is familiar with the architecture as outlined herein in an expedited manner or automatically as described below in detail.

Layer **1** can be precisely defined in generic terms, and is therefore the simplest layer to likely be automatically, substantially automatically, or easily manually generated. A sample or example of actual code that can be used to implement this layer is described below. As long the process and/or code for implementing Layer **1** can be generically defined, that is engine and technology independent, then the process of generating the generic code for each new engine is expedited either manually or automatically.

The premise for automating any level is to start with as few pieces of information as possible. For the Engine

Management layer I have assumed that nothing more than the set of DLLs that implement the engine functionality are known. Given this information, I have determined that I will need to implement:

Loading and unloading the engine from memory

Adding error management

We can start, in this example, with a model C++ header file that defines the Engine Management layer and investigate how this code can be implemented generically. As mentioned earlier, if the code to implement this layer can be defined generically then it can be easily generated, for example, manually, and/or automatically for any engine.

```
class SomeEngineObject
{
    //Wrapper Functions
private:
    FARPROC_SomeFunction;
    BOOL SomeFunction0;
    //EngineManagement
protected:
    BOOL GetProcAddress (HINSTANCE, FARPROC&, LPCTSTR);
    BOOL GetProcAddresses0;
    BOOL ProcessError0;
public:
    BOOL ActivateEngine (BOOL Activate);
    BOOL IsEngineActivated0;
};
```

The IEngineManagement interface is implemented in the C++ class as the public methods: ActivateEngine( ) and IsEngineActivated( ).

The first step of implementing the Engine Management layer **20** is to wrap each original engine function within a class-defined function that represents the original. For example, if there is an original function called Somefunction( ), then the engine object should have a corresponding Somefunction( ) method. The engine object version can then add standard engine and error management code so that any layers above have automatic error detection, correction, and reporting.

An example of generic code that maps an original function call to the original function is as follows:

```
BOOL GetProcAddress (HINSTANCE hLib, FARPROC&Proc,
LPCTSTR ProcName)
{
    Proc=::GetProcAddress (hLib, ProcName)
    if (!Proc)
    {
        SetIMAGmEError (LOADENGINEFUNCTIONSERROR,
ProcName);
        return FALSE;
    }
    return TRUE;
```

Given the original function name, the GetProcAddress can map the original function to one that is defined by the engine object. Using the engine object C++ header file described above, the Somefunction( ) method is mapped to the original engine function using the following line of code:

```
(GetProcAddress(hLib, SomeFunction, "SomeFunction" );
```

To map all the function calls within the original engine DLLs just requires cycling through each function call and mapping it to the engine object counterpart. Since Windows contains facilities that enables access to all the functions within a DLL, a simple loop may be used. The hLib module is derived from the DLL name, which, as mentioned at the start, is the one piece of information we are given.

What is more complex is to define a generic implementation of the engine object version of the original function. This may be described in code as follows:

```
    BOOL SomeFunction (arguments)
    ASSERT (arguments)
    ErrorVariable=_SomeFunction (arguments);
    returnProcessError0;
}
```

The engine object version of the original function passes the function call to the original one after completing a series of assertion tests, and is followed by a series of error detection tests. In this way the original engine function is "wrapped" by the engine object to manage error detection and correction.

The process of loading an engine can likewise be implemented generically.

```
BOOLLoadDLLs0
{
    BOOLbReturn=TRUE;
    HINSTANCEt_hLib;
    CStringt_ModuleName;
    POSITIONpos;
    pos=m_Modules · GetStartPosition0;
    if (pos=NULL)
    {
        SetIMAGinEError (NOMODULESDEFINED);
        return FALSE;
    }
    while (pos&&bReturn)
    {
        m_Modules · GetNextAssoc (pos, t ModuleName, t hLib);
        if (t_hLib!=NULL)
            continue;
        t_hLib=::LoadLibrary (t_ModuleName);
        if (t_hLib=NULL)
        {
            SetIMAGinEError (CANTLOADMODULE,
            t_ModuleName);
            FreeDLLs0;
            bReturn=FALSE;
            break;
        }
        m_Modules · SetAt (t_ModuleName, t_hLib);
    }
returnbReturn;
}
```

The LoadDLLs function is a generic implementation of a function that loops through the names of DLLs that are provided (in the form of the m_Modules variable), and cycles through each one loading it into memory using the Windows LoadLibrary( ) function. A similar engine object function can be implemented to remove these DLLs from memory.

The present invention further divides the engine management layer into three functions, as illustrated in FIG. **8**. The first function is loading and unloading **124** of the core or engine technology. The second function for the engine management layer **26** is dynamically linking procedures or function calls, or hooking the desired engine functionality into the procedures of the core technology **126**, including, for example, initializing and setting up engine settings. The third function is initializing the engine itself **128**, which is essentially engine management. Once these three functions are performed in level 1, anything in the core technology is accessible.

Advantageously, the present invention utilizes tables to drive each of these three functions described above, and as illustrated in FIG. **9**. Each of the tables of files, for example tables **130**, **136**, **140**, are filled in with the appropriate data or information. I have discovered that if the above three functions are set up or implemented using tables, that the core technology may be effectively and efficiently described. That is, the use of tables is a very effective and simple method of describing an engine for use in engine management, engine loading/unloading and engine procedure linking. For example, it is similar to indicating or providing the raw data of that engine, the list of the engine functions, and the list of the engine dynamic link libraries (DLLs) for engine management.

The files or tables contain the logic or executable of the engine. Accordingly, all that is needed is a list of the engine functions **132**, a list of the file of the engine executable code or DLLs **138**, and a list of the engine settings **142**. Using the tables with the above information, the engine may be automatically loaded and unloaded, initialized, and/or dynamically hooked into the necessary functions. Accordingly, the process of generating level 1 for engine management may advantageously be automated. The specific algorithms used for the engine management layer are described in Appendix A.

In summary, for the Engine Management layer the following pieces may be automated, substantially automated, and/or manually expedited.

- Loading and unloading the engine DLLs (provided into and out of memory
- Mapping original functions to engine object counterparts
- Adding general error detection and correction
- Determining and matching arguments and return values for mapping the original functions to their engine object counterparts. In order to add assertion and error detection and correction, the original function must be wrapped and called from within the engine object version of the original function.
- Managing error feedback. All APIs have their own way of providing error feedback. Since one of the goals of the Engine Management layer is to generically manage error detection, correction, and feedback, it must handle all errors identically. However, APIs have numerous and incompatible methods in this case. I have determined that most APIs follow one of several distinct mechanisms for providing error feedback. By creating specific classes of APIs, the process of generating Layer **1** engine management may be expedited, manually and/or automatically.

Layer **2**—Engine Configuration **24**

The second layer **24** in the object component architecture is designed to deal with configuring an engine. This includes the ability to set any variety of features that are generally associated with the functioning of an engine. The architecture is designed to meet the challenge of providing a uniform interface for dealing with generally any or most engine settings.

The engine configuration layer **24** includes a series of prefabricated functions that map out the settings stored in the table to the appropriate engine configuration parameters. Accordingly, all that is needed is to fill in the values for the table associated with engine configuration. Thus, the engine object may advantageously come pre-packaged with predetermined tables populated with predetermined values.

The Layer **2** specification can be summarized in FIG. **10** that describes an exemplary IEngineConfiguration COM interface. The purpose of the IEngineConfiguration interface is to provide the ability to set and get the settings of any engine uniformly. While the Engine Management layer can load and unload engines transparently, this layer configures engines to operate as required by the user or developer.

FIG. **11** is another exemplary table illustrating the engine configuration specification. Examples include a set setting function **144**, a get setting function **146**, a load setting **148**, a save setting **150**, an is setting valid function **152**, a default setting **154**, and a prompt setting **156**.

The get setting **146** and set setting **144** functions retrieve the value of a particular engine setting, or assign a value to a particular engine setting, respectively. Each one of the get setting and set setting functions includes or comprises a table of the settings. The load setting **148** and save setting **150** functions do the similar function as the get setting and set setting functions, but in persistence. Persistence is defined as writing values to the disk, for example hard disk, compact disk, and the like, and retrieves the values from the disk. So as where the get setting and set setting functions assign a value and/or retrieves the value from local memory, the load and set setting functions assign the value and retrieve the value of the setting from disk. The load and set setting functions provide persistence when the computer system is close down, such that when the computer system will return to the last setting when it is subsequently reopened.

The default setting function **154** provides the most favorable value for a given setting. Thus, if no setting is selected, the system will automatically select default settings. The prompt setting function **156** is what displays to the user all the various settings. The specific algorithms used to implement the engine configuration layer are included in Appendix B.

Advantageously, the present invention generates the skeletal structure of each table automatically. In addition, since there is a table of settings, the skeletal structure not only generates these functions, but also fills in the settings that need to be assigned. Thus, the engine configuration function provides the feature of having a pre-populated set of options which require particular values to be assigned to table entries.

Although this architecture advantageously makes it simple for a human to migrate the configuration of an engine appear into two simple and universally applicable interface points, doing so automatically requires additional steps. The two steps to automating this approach are, for example, as follows:

- Determine the configuration methods used by various APIs for configuring the core technology;
- Detect the variations for configuring an engine and automating each one separately.

As with Layer **1**—Engine Management, there exists a finite set of general variations used by developers of core technologies to configure an engine. Although Layer **1** is clearly more generic in nature, advantageously, Layer **2** also has considerable consistency.

Layer **3**—Engine Functionality **22**

The third layer **22** in the object component architecture is designed to deal with accessing the actual functionality of the core engine. For example, for an OCR engine this would be to OCR an image or a document. For a text retrieval engine this would be to initiate and retrieve results of a text search.

An exemplary Layer **3** specification can be summarized in FIG. **12** that describes the IEngineFunction COM interface. The purpose of the IEngineFunction interface is to provide

the ability to initiate any function supported by an engine. The simple IEngineFunction interface is capable of managing an infinite variation of functions.

The third layer may advantageously be further divided into many sub-layers that more discretely define the steps necessary to execute a function within an API. Since the designer of an API has infinite variety of possible ways of implementing a function, creating a tiered architecture to manage this layer is useful.

An exemplary tiered architecture for the engine function is illustrated in FIG. **13**. As illustrated in FIG. **13**, the engine function or engine processing layer includes four elements. The engine function layer **22** includes a series of predefined functions to perform in the perform element **158**. For example, for optical character recognition (OCR), the present invention uses a set of predefined functions. Alternatively, for scanning, the present invention includes a separate set of predefined functions.

Accordingly, there are a series of actions that are performed by the engine function layer on a given engine, such as an OCR engine, a scanning engine, a printing engine and the like. The engine function layer is designed not to generally go directly to a specific engine. Rather, the engine function layer **22** will generally interface with the engine management layer **26** and/or the engine configuration layer **24** as needed.

For example, in the course of performing an action and/or function, the engine function layer interfaces with the engine configuration layer to possibly modify settings. For an OCR engine, the engine function layer fills out a table of OCR documents as one action that could take place. OCR image is another action.

The get function results **160** gets the results of the function stored in a register. The clear function **162** clears all the registers that contain all the results, in this case its memory. The feedback event or function **164** provides continuous feedback, depending on what action takes place. For example, if an OCR action is being performed, the feedback function provides the percentage of completion of the OCR process. The specific algorithms used to implement the engine function layer are included in Appendix C.

The automation of this layer is accomplished by the following functions:

Determine the execution of methods used by various APIs for executing a given function;

Divide this layer into a multi-tiered layer that further facilitates automation;

Detect the variations of the sub-layers and automate each one separately.

Although this layer has many more variations than Layer **2**, I have determined that there is a general set of variations used by developers of APIs to implement core functionality.

Thus, the benefit of the component factory is that it can transform core software technologies that are currently available in "C"-level APIs to a limited audience into components that have a much greater audience.

There are a variety of "C"-level APIs that cover the following categories of functionality that can be better served in the market as ActiveX controls or other component and used in conjunction with the architecture and methods described herein.

Text Retrieval

Data Extraction

Workflow

Storage Management

Each of these categories has several vendors with products that currently service the market in a limited way because the technologies are only available as "C"-level APIs. Without the core competency of creating components out of these core technologies they are limiting their marketability and opportunity for international distribution.

With the proposed component factory users and vendors can rapidly create components from their original core technology and increase their marketability, competitiveness, and ultimately their sales.

Further, there are numerous core technologies, such as text-retrieval and ICR (Intelligent Character Recognition), that have already been implemented, and are only available as "C"-level APIs. Many, if not most, core technologies are first released exclusively as "C"-level APIs. While there are integrators and corporations who have the team of technologists who can integrate these "C"-level APIs in-house, most companies are looking for component versions that can be implemented at a much higher level. Therefore, many of the core technologies that are only available in a "C"-level API are not being used due to their inaccessible interface. The benefit of the component factory is that it can rapidly make available core technologies implemented as "C" APIs that would otherwise be underutilized or dormant in research labs by converting them to high-level components that can be used by millions of power-PC users.

With the advent of the World Wide Web (WEB) this opportunity has increased exponentially. The WEB is now home to a vast number of WEB authors with minimal formal training who can implement HTML pages and build web sites. One of the fundamental technologies for extending the capability of the WEB from simple page viewing to interactive and sophisticated applications is components. A component extends the capability of HTML by enabling a WEB author to add core technology as a pre-packaged technology. Since components are fundamental to the growth and usability of the WEB, having a component factor that can translate "C"-level toolkits into components that are then usable within WEB sites opens a vast and new worldwide market to these technologies.

FIG. **14** is an illustration of a main central processing unit for implementing the computer processing in accordance with a computer implemented embodiment of the present invention. The procedures described above may be presented in terms of program procedures executed on, for example, a computer or network of computers.

Viewed externally in FIG. **14**, a computer system designated by reference numeral **40** has a central processing unit **42** having disk drives **44** and **46**. Disk drive indications **44** and **46** are merely symbolic of a number of disk drives which might be accommodated by the computer system. Typically these would include a floppy disk drive such as **44**, a hard disk drive (not shown externally) and a CD ROM indicated by slot **46**. The number and type of drives varies, typically with different computer configurations. Disk drives **44** and **46** are in fact optional, and for space considerations, may easily be omitted from the computer system used in conjunction with the production process/apparatus described herein.

The computer also has an optional display **48** upon which information is displayed. In some situations, a keyboard **50** and a mouse **52** may be provided as input devices to interface with the central processing unit **42**. Then again, for enhanced portability, the keyboard **50** may be either a limited function keyboard or omitted in its entirety. In addition, mouse **52** may be a touch pad control device, or a track ball device, or even omitted in its entirety as well. In addition, the computer system also optionally includes at least one infrared transmitter **76** and/or infrared receiver **78**

for either transmitting and/or receiving infrared signals, as described below.

FIG. **15** illustrates a block diagram of the internal hardware of the computer of FIG. **14**. A bus **56** serves as the main information highway interconnecting the other components of the computer. CPU **58** is the central processing unit of the system, performing calculations and logic operations required to execute a program. Read only memory (ROM) **60** and random access memory (RAM) **62** constitute the main memory of the computer. Disk controller **64** interfaces one or more disk drives to the system bus **56**. These disk drives may be floppy disk drives such as **70**, or CD ROM or DVD (digital video disks) drive such as **66**, or internal or external hard drives **68**. As indicated previously, these various disk drives and disk controllers are optional devices.

A display interface **72** interfaces display **48** and permits information from the bus **56** to be displayed on the display **48**. Again as indicated, display **48** is also an optional accessory. For example, display **48** could be substituted or omitted. Communication with external devices, for example, the components of the apparatus described herein, occurs utilizing communication port **74**. For example, optical fibers and/or electrical cables and/or conductors and/or optical communication (e.g., infrared, and the like) and/or wireless communication (e.g., radio frequency (RF), and the like) can be used as the transport medium between the external devices and communication port **74**.

In addition to the standard components of the computer, the computer also optionally includes at least one of infrared transmitter **76** or infrared receiver **78**. Infrared transmitter **76** is utilized when the computer system is used in conjunction with one or more of the processing components/stations that transmits/receives data via infrared signal transmission.

FIG. **16** is a block diagram of the internal hardware of the computer of FIG. **14** in accordance with a second embodiment. In FIG. **16**, instead of utilizing an infrared transmitter or infrared receiver, the computer system uses at least one of a low power radio transmitter **80** and/or a low power radio receiver **82**. The low power radio transmitter **80** transmits the signal for reception by components of the production process, and receives signals from the components via the low power radio receiver **82**. The low power radio transmitter and/or receiver **80**, **82** are standard devices in industry.

FIG. **17** is an illustration of an exemplary memory medium which can be used with disk drives illustrated in FIGS. **14–16**. Typically, memory media such as floppy disks, or a CD ROM, or a digital video disk will contain, for example, a multi-byte locale for a single byte language and the program information for controlling the computer to enable the computer to perform the functions described herein. Alternatively, ROM **60** and/or RAM **62** illustrated in FIGS. **15–16** can also be used to store the program information that is used to instruct the central processing unit **58** to perform the operations associated with the production process.

Although processing system **40** is illustrated having a single processor, a single hard disk drive and a single local memory, processing system **40** may suitably be equipped with any multitude or combination of processors or storage devices. Processing system **40** may, in point of fact, be replaced by, or combined with, any suitable processing system operative in accordance with the principles of the present invention, including sophisticated calculators,and hand-held, laptop/notebook, mini, mainframe and super computers, as well as processing system network combinations of the same.

Conventional processing system architecture is more fully discussed in *Computer Organization and Architecture,* by William Stallings, MacMillam Publishing Co. (3rd ed. 1993); conventional processing system network design is more fully discussed in *Data Network Design,* by Darren L. Spohn, McGraw-Hill, Inc. (1993), and conventional data communications is more fully discussed in *Data Communications Principles,* by R. D. Gitlin, J. F. Hayes and S. B. Weinstain, Plenum Press (1992) and in *The Irwin Handbook of Telecommunications,* by James Harry Green, Irwin Professional Publishing (2nd ed. 1992). Each of the foregoing publications is incorporated herein by reference. Alternatively, the hardware configuration may be arranged according to the multiple instruction multiple data (MIMD) multiprocessor format for additional computing efficiency. The details of this form of computer architecture are disclosed in greater detail in, for example, U.S. Pat. No. 5,163,131; Boxer, A., Where Buses Cannot Go, IEEE Spectrum, February 1995, pp. 41–45; and Barroso, L. A. et al., RPM: A Rapid Prototyping Engine for Multiprocessor Systems, IEEE Computer February 1995, pp. 26–34, all of which are incorporated herein by reference.

In alternate preferred embodiments, the above-identified processor, and in particular microprocessing circuit **58**, may be replaced by or combined with any other suitable processing circuits, including programmable logic devices, such as PALs (programmable array logic) and PLAs (programmable logic arrays). DSPs (digital signal processors), FPGAs (field programmable gate arrays), ASICs (application specific integrated circuits), VLSIs (very large scale integrated circuits) or the like.

FIG. **18** is an illustration of another embodiment of the component factory migrating the original "C"-level API from its original state into the generic interface defined by the topmost layer. This powerful architecture goal is to supply easy access to all imaging functions that can be performed by any engine.

The architecture according to this second embodiment, groups C-level toolkits **100** into logical categories, such as scan, print, display, OCR, cleanup and so on. A single engine can span multiple categories (e.g., Kofax engine does view/print/scan). This enables the architecture to deal with the multitude of engines available in a logical fashion.

On top of these, a three-level C++ class (or object) **102** is built for each engine. This object gives uniform access to the engine and to all its unique settings. The three levels do the following:

Level 1 of the C++ classes **112** is a protective wrapper for each function call in the underlying engine. It traps all errors and provides error management and administration to prevent accidental GPFs or engine crashes.

Think of it as the "condom layer." While providing the most direct access feasible to the underlying engine and all its capabilities, level 1 of the C++ class **112** also protects the user from the engine. It manages all engine loading and unloading, prevents multiple copies of an engine and calls engines automatically as needed.

The architecture also provides three levels of access: 1. Use the default engine settings. Benefit: No learning up front. Program knowing nothing other than "OCR gets text out of there." 2. Prepackage customized engine settings. Set it once for everyone who uses the program, every time they use the program. 3. Modify engine settings at run-time. Let the user choose the settings.

Level 2 of the C++ classes **114** bridges the low-level API calls so they can be used by level 3 **116** in standardized calls for each category. And it supplements the engine by pro-

viding additional functionality, such as safely loading and unloading engines.

Level 3 of the C++ class **116** consists of a standardized set of calls for all engines in each category. Programmers can access all the unique functions of each engine in a uniform way.

Another associated C++ class, called a Visual Class **104**, adds a visual interpretation of each engine. This class manages all user interaction with each underlying engine. Like their lower-level counterparts, the Visual Class consists of three layers:

Level 1—**118** adds any dialogs or other pop-up window capability that may be lacking in each engine. Examples: Dialogs to customize the engine settings or, for a recognition engine, the zone definition settings.

Level 2—**120** serves two functions: It bridges level 1 dialogs with the actual Windows window that represents the control. It also handles all Windows-related error message presentation.

Level 3—**122** manages anything else from the underlying engine (such as annotations) that needs to appear on the window. The Visual Class includes engine-specific Windows dialog boxes that let you customize which engine features you want to use, as well as any other Windows representation necessary for a toolkit. (For example, a compression engine has to display the image—the visual class, not the engine, does the work.)

The Object Manager layer **106**, the first horizontal umbrella, orchestrates the underlying objects. It translates service requests into a form that the engine objects can understand.

The Windows Manager **108** presents Windows messages (move window, mouse/scrollbar/toolbox activity) to the Object Manager. It is written using Microsoft's Foundation Class (MFC), which makes it easy to support OCXs. (The OCX is in fact an MFC class.)

At the top, a visual interface **110** presents to the user a set of visual calls and translates those calls into Windows messages. This layer comprises only 5% of the VBX code, yet it permits the toolkit to appear as a VBX, OCX or other standard visual interface.

Accordingly, the present invention provides two main layers, the engine object component layer and the object manager layer. By creating these two main layers, the present invention allows third parties to create their own engine object component layers so that the third party engine can be readily compatible and useable by the present invention. In addition, the present invention is accessible via the Internet. That is, the present invention is operable over the Internet using, for example, standard Internet protocols, such as component object module (COM) communication protocol and distributed COM (DCOM) protocol.

In addition, the present invention optionally combines three layers of functions including the visual interface, the windows manager and the object manager into one layer called the object manager. Of course, this combination of layers is not meant to convey that only these specific layers must be used, but rather, to be indicative of overall functionality generally required to implement or execute component engines. That is, one or more of the above functions may be incorporated into the object manager layer. The present invention also advantageously combines the visual classes and C++ classes into the engine object component to further standardize and/or provide access to the object manager for engine object components.

The present invention optionally uses the standard ActiveX component control supplied, for example, by MicroSoft Corporation. ActiveX is a protocol for component communication. The present invention also creates each of the object manager and the engine component layer as a separate ActiveX. That is, the object manager is its own ActiveX control, and the engine object is its own ActiveX control. Thus, the engine object can now run independently from the object manager. Accordingly, the engine object can operate without relying necessarily on the concurrent operation of the object manager.

The independent relationship between the engine object and the object manager means also that the engine object represents a discrete means of technology. For example, an engine object can be an OCR technology. This provides several benefits. First, because the object manager layer is open, the manufacturer of the OCR technology can wrap their own engine in the form of an engine object component, and the engine will automatically "plug into" or work with, the object manager. Thus, the engine object is provided high level access, making it accessible to many more parties, users, and the like. When the object manager interface is designed to be open, any third party, such as an engine manufacturer, can create their own engine object component that is compatible with the object manager, the manufacturer can do it.

FIG. **19** is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for client server and/or intranet operating environments. A very significant point that is relevant to why the object manager and the engine object component are independent in the present invention relates to providing a distributed environment for using the present invention. Rather than communicate within the same technology between the object manager and the engine object, the object manager and the engine object component communicate with each other in binary mode, via, for example, standard distributed component object module (DCOM) communication. As illustrated in FIG. **19**, object manager **14** communicates with engine object component **16, 18, 20** via DCOM specification **166**. Other types of component communication may also be utilized that provide the capability of a distributed component interaction.

Thus, the engine object component and the object manager can leverage current protocols to not only communicate on the same machine, but also on different machines such as a client server and/or intranet and/or Internet environment. The object manager can be placed on one machine, and the engine object component on another machine and have distributed processing, what is otherwise called thin client processing, distributed processing, wide area intranet processing.

What this allows the present invention to do is to put the object manager on the thin client, who would accept the request from the user, for example, to OCR something or to print something. The actual request is handled or processed by the engine object component which generally resides on the server. The engine object component contains the horse power, or the processing power to process the request.

The engine object layer is generally located in the same or substantially same location as where the core technology or engine itself is being stored. Alternatively, the engine object layer and the engine may be optionally located in a distributed environment on different machines, servers, and the like.

FIG. **20** is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for

client server and/or intranet operating environments. In FIG. **20**, client **170** includes object manager layer **172**. Client **170** executes the core technology **180**, via accessing engine object layer **178** managed/stored on server **176**, and communicated via server **174**.

Client **182**, located on the same server **176** as core technology **180** and engine object layer **178**, may also be used to execute the core technology **180** via object manager layer **184**. In this instance, the client **182** with the object manager layer **184** is located on the same server **176** as the engine object layer **178**. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine object component layer and the object manager layer on the same machine **186**, as well as the core technology.

Further, since the object manager is formatted or constructed of a client technology, the object manager can sit in a standard browser. This means that anyone that has an Internet browser, i.e., anyone that has access to the world wide web (WEB) can actually access the core engine technology. Thus, by structuring the architecture of the present invention as described herein, users automatically become Internet, intranet and/or WEB enabled.

The present invention also transforms the core technology from essentially client based technology into a client server and/or a thin client technology. This makes the core technology high level accessible, thereby transforming any core technology into client server, or hidden client technology. The browser is located on the client, and the browser leverages the object manager. Accordingly, the browser optionally contains the object manager, and the object manager makes requests over, for example, the Internet, local network, and the like via a server, to the engine object. The server would be either a web server or a LAN server.

The present invention also advantageously provides the ability to have the client and the server, in a distributed environment as discussed above, or on the same machine locally. The present invention utilizes the DCOM communication protocol defining the communication protocol between the object manager and the engine object component. Accordingly, since DCOM can work on the same machine as well as in a distributed environment, DCOM does not necessitate that the engine object or the object manager component be on two separate machines.

FIG. **21** is an illustration of a distributed environment or architecture for manually and/or automatically generating and/or using reusable software components for network environments, such as the Internet. As illustrated in FIG. **21**, object manager **14** communicates with engine object component **16, 18, 20** via DCOM specification and a networking environment, such as the Internet, intranet, and the like **168**. Other types of component communication may also be utilized that provide the capability of a distributed component interaction over a networking environment.

FIG. **22** is a detailed illustration of the distributed environment or architecture for manually and/or automatically generating and/or using reusable software components in the Internet environment. In FIG. **22**, client **170** includes object manager layer **172**. Browser/thin client **170** a executes the core technology **180**, via accessing engine object layer **178** managed/stored on web server **176*a***, and communicated via the Internet **174*a***.

Browser/thin client **182*a***, located on the same web server **176*a*** as core technology **180** and engine object layer **178**, may also be used to execute the core technology **180** via object manager layer **184**. In this instance, the browser/thin client **182*a*** with the object manager layer **184** is located on the same web server **176*a*** as the engine object layer **178**. In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine object component layer and the object manager layer on the same machine **186**, as well as the core technology.

FIGS. **23A–23C** are illustrations of the image viewer user selectable or configurable or programmable interface and/or functionality associated therewith in accordance with the present invention. In FIG. **23A**, user interface **200** for image viewing includes viewing frame **202**, with dual viewing areas **204, 206**. Viewing area **204** includes at the periphery, previous page activator **208**, at the top, document tools **210**, and at the bottom status indicator **214**. Viewing area **206** includes at the periphery, next page activator **212**, at the top, document tools **214**, and at the bottom status indicator **216**.

Advantageously, this user interface is selectable and/or customizable by the user, as illustrated below in connection with this figure and FIGS. **23B–23C**. Significantly, the image viewer provides the ability to a user to retain or develop a specific perspective on viewing a document. One of the features of the viewer is therefore the ability to change the user's perspective. For example, the user might be looking at the same document, as a book, as a film, or as a bounded or traditional book. This gives the user the ability to relate to the document in a fashion that they are comfortable with, depending on the content or depending on the user. That is, the image viewer is like a usable selectable perspective on viewing a document in a plurality of ways.

FIG. **23B** is an illustration of another user selectable interface for image viewing. In FIG. **23B**, user interface **200'** for image viewing includes viewing frame **202'**, with single viewing area **204'**. Viewing area **204'** includes at the top left, previous page activator **208'** and at the top right next page indicator **212'**. Viewing area **204'** also includes at the left area document tools **210'**, and at the bottom status indicator **214'**. Viewing area **204'** also includes at the top, multiple viewing page area **218**, that appears and preferably moves like a film, and provides viewing of multiple consecutive or non-consecutive pages. Advantageously, this user interface is selectable and/or customizable by the user, as illustrated below in connection with this figure and FIG. **23A** and FIG. **23C**.

FIG. **23C** is an illustration of another user selectable interface for image viewing. In FIG. **23C**, user interface **200"** for image viewing includes viewing frame **202"**, with single viewing area **204'**. Viewing area **204"** includes at the top right, previous page activator **208"** and at the bottom left next page indicator **212"**. Viewing area **204"** also includes at the left area document tools **210"**, and at the bottom status indicator **214"**. Viewing area thus provides a user interface to view a document that appears like a bound or more traditional book. Advantageously, this user interface is selectable and/or customizable by the user, as illustrated below in connection with this figure and FIGS. **23A–23B**.

FIG. **24** is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments. The architecture in FIG. **24** provides the capability to perform the viewer process off-line. That is, the viewer process **188** provides an added feature on top of the object manager layer **14**. As described above, object manager layer **14** is essentially an interface, and the viewer process **188** is an application that leverages the object manager layer **14**.

The advantage of the viewer process **188** being built on the object manager layer **14**, which is built on top of the

engine object layer **16, 18, 20,** is that the viewer process can offset its processing capabilities anywhere in a distributed environment. It can have the processing occur at the local station, on a server, and the like, as described below in detail. Significantly, the object manager and the engine object component are independent to provide a distributed environment for using the present invention. Rather than communicate within the same technology between the object manager and the engine object, the object manager and the engine object component communicate with each other in binary mode, via, for example, standard distributed component object module (DCOM) communication.

As illustrated in FIG. **24,** object manager **14** communicates with engine object component **16, 18, 20** via DCOM specification **166.** Other types of component communication may also be utilized that provide the capability of a distributed component interaction. Object manager **14** is also respectively connectable to viewer process **188.**

Thus, the engine object component and the object manager can leverage current protocols to not only communicate on the same machine, but also on different machines such as a client server and/or intranet and/or Internet environment. The object manager and/or viewer process can be placed on one machine, and the engine object component on another machine and have distributed processing, what is otherwise called thin client processing, distributed processing, wide area intranet processing.

What this allows the present invention to do is to put the object manager on the thin client, who would accept the request from the user, for example, to perform the viewer process. The actual request is handled or processed by the engine object component which generally resides on the server. The engine object component contains the horse power, or the processing power to process the request.

The engine object layer is generally located in the same or substantially same location as where the core technology or engine itself is being stored. Alternatively, the engine object layer and the engine may be optionally located in a distributed environment on different machines, servers, and the like.

FIG. **25** is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in client server and/or intranet operating environments. In FIG. **25,** client **170** includes object manager layer **172** with viewer process **192.** Client **170** executes the core technology **180,** via accessing engine object layer **178** managed/stored on server **176,** and communicated via server **174.** Viewer process **190** is also optionally available to either or both servers **174, 176.**

Client **182,** located on the same server **176** as core technology **180** and engine object layer **178,** may also be used to execute the core technology **180** and/or viewer process **192** via object manager layer **184.** In this instance, the client **182** with the object manager layer **184** is located on the same server **176** as the engine object layer **178.** In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine object component layer, viewer process **194** and the object manager layer on the same machine **186,** as well as the core technology.

Further, since the object manager is formatted or constructed of a client technology, the object manager can sit in a standard browser. This means that anyone that has an Internet browser, i.e., anyone that has access to the world wide web (WEB) can actually access the core engine technology and/or viewer process. Thus, by structuring the architecture of the present invention as described herein, users automatically become Internet, intranet and/or WEB enabled.

The present invention also transforms the core technology and/or viewer process from essentially client based technology into a client server and/or a thin client technology. This makes the core technology high level and/or viewer process accessible, thereby transforming any core technology and./or viewer process into client server, or hidden client technology. The browser is located on the client, and the browser leverages the object manager. Accordingly, the browser optionally contains the object manager, and the object manager makes requests over, for example, the Internet, local network, and the like via a server, to the engine object. The server would be either a web server or a LAN server.

The present invention also advantageously provides the ability to have the client and the server, in a distributed environment as discussed above, or on the same machine locally. The present invention utilizes the DCOM communication protocol defining the communication protocol between the object manager and the engine object component. Accordingly, since DCOM can work on the same machine as well as in a distributed environment, DCOM does not necessitate that the engine object or the object manager component be on two separate machines.

FIG. **26** is an illustration of a stand-alone and/or distributed environment or architecture for image viewer in network environments, such as the Internet. As illustrated in FIG. **21,** object manager **14** communicates with engine object component **16, 18, 20** via DCOM specification and a networking environment, such as the Internet, intranet, and the like **168.** In addition, object manager layer **14** also advantageously communications with viewer process **188***a.* Other types of component communication may also be utilized that provide the capability of a distributed component interaction over a networking environment.

FIG. **27** is a detailed illustration of a stand-alone and/or distributed environment or architecture for image viewer in the Internet environment. In FIG. **27,** client **170** includes object manager layer **172.** Browser/thin client **170***a* executes the core technology **180** and/or viewer process **192***a,* via accessing engine object layer **178** managed/stored on web server **176***a,* and communicated via the Internet **174***a.* Viewer process **190** is also optionally available to web server **176***a.*

Browser/thin client **182***a,* located on the same web server **176***a* as core technology **180,** viewer process **192***a* and engine object layer **178,** may also be used to execute the core technology **180** via object manager layer **184.** In this instance, the browser/thin client **182***a* with the object manager layer **184** is located on the same web server **176***a* as the engine object layer **178.** In addition, since the present invention utilizes a communication protocol between components, for example, DCOM, that allows a client to also include both the engine object component layer and the object manager layer on the same machine **186,** as well as the core technology and viewer process.

The many features and advantages of the invention are apparent from the detailed specification, and thus, it is intended by the appended claims to cover all such features and advantages of the invention which fall within the true spirit and scope of the invention. Further, since numerous modifications and variations will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation illustrated and described, and accordingly, all suitable modifications and equivalents may be resorted to, falling within the scope of the invention.

For example, while the above discussion has separated the various functions into separate layers of functionality, the layers may be combined, physically and/or logically, and various functions may be combined together. While combining various functions into same or common layers may make implementation details more cumbersome, nevertheless, the functions described herein may still be accomplished to advantageously provide some or all of the benefits of the invention described herein.

Further, as indicated herein, the present invention may be used to automate and/or manually expedite the migration of a program specific Application Programmer Interface from an original state into a generic interface by building an object for each engine. The object advantageously provides substantially uniform access to the engine and engine settings associated with the engine. The present invention may be applied across a broad range of programming languages that utilize similar concepts as described herein.

What is claimed is:

1. A distributed computer implemented process for migrating at least one program specific Application Programmer Interface (API) from an original state into a substantially consistent interface by building an object for at least one of an engine and a viewer process, the object providing substantially uniform access to the at least one of the engine having engine settings and the viewer process, comprising the steps of:

(a) providing, on a server, the at least one engine and viewer process, each with one or more features to be executed;

(b) providing, on at least one of the server and another server connectable to the server, at least one engine component or another viewer process configured to execute the one or more features by converting the at least one program specific Application Programmer Interface (API) from the original state into the substantially consistent interface, and mapping the substantially consistent interface to the at least one of the engine and the viewer process; and

(c) providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an object manager layer communicable with and managing the at least one engine component or the another viewer process via the substantially consistent interface.

2. A distributed computer implemented process according to claim **1**, wherein the object manager layer is consistently communicable with each engine or the viewer process using the at least one engine component via the substantially consistent interface.

3. A distributed computer implemented process according to claim **1**, wherein the substantially consistent interface comprises at least one of a component object module (COM) communication protocol and a distributed COM (DCOM) protocol.

4. A distributed computer implemented process according to claim **3**, wherein the COM protocol comprises an ActiveX control.

5. A distributed computer implemented process according to claim **1**, wherein the substantially consistent interface comprises a binary mode communication protocol.

6. A distributed computer implemented process according to claim **5**, wherein the binary mode communication protocol includes a distributed component object module (DCOM) protocol.

7. A distributed computer implemented process according to claim **5**, wherein the binary mode communication protocol provides the capability for the engine component to execute independent of the object manager layer.

8. A distributed computer implemented process according to claim **1**, wherein the client comprises at least one of a browser and a thin client, and the server comprises a web server.

9. A distributed computer system for migrating at least one program specific Application Programmer Interface (API) from an original state into a substantially consistent interface by building an object for at least one of an engine and a viewer process, the object providing substantially uniform access to the at least one of the engine having engine settings and the viewer process, comprising:

a server configured to include

the at least one engine and viewer process, each with one or more features to be executed;

at least one engine component or another viewer process configured to execute the one or more features by converting the at least one program specific Application Programmer Interface (API) from the original state into the substantially consistent interface and mapping the substantially consistent interface to the at least one of the engine and the viewer process; and

a client configured to be connectable to said server and optionally configured to be connectable to another server, said client including an object manager layer communicable with and managing the at least one engine component or the another viewer process via the substantially consistent interface.

10. A distributed computer system according to claim **9**, wherein the object manager layer is consistently communicable with each engine using the at least one engine component via the substantially consistent interface.

11. A distributed computer system according to claim **9**, wherein the substantially consistent interface comprises at least one of a component object module (COM) protocol and a distributed COM (DCOM) protocol.

12. A distributed computer system according to claim **11**, wherein the COM protocol comprises an ActiveX control.

13. A distributed computer system according to claim **9**, wherein the substantially consistent interface comprises a binary mode communication protocol.

14. A distributed computer system according to claim **13**, wherein the binary mode communication protocol includes a distributed component object module (DCOM) protocol.

15. A distributed computer system according to claim **9**, wherein the client comprises at least one of a browser and a thin client, and the server comprises a web server.

16. An image viewer process for viewing at least one document image including an electronic document image, and for performing viewing operations to the electronic document image by a user, comprising the steps of:

(a) selecting, by the user, one of a plurality of image viewing perspectives, each of the plurality of image viewing perspectives providing the user the capability of viewing the document image in accordance with a different predefined user perspective;

(b) selecting, by the user, using the image viewer process the document image to be viewed;

(c) retrieving, by the image viewer process, the document image;

(d) providing by at least one of the user and a computer at least one substantially consistent program specific Application Programmer Interface (API) for the at least two processing application; and

(e) displaying, by the image viewer process, the document image selected by the user in said selecting step (b), using the at least one substantially consistent API provided in said providing step (d), in accordance with the one of a plurality of imaging viewing perspectives selected by the user in said selecting step (a).

17. A distributed computer implemented process including an object providing substantially uniform access to at least one engine having engine settings associated with the engine and an engine interface for accessing the engine settings and one or more features to be executed, comprising the steps of:

(a) providing, on at least one of the server and another server connectable to the server, at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine; and

(b) providing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an object manager layer communicable with and managing the at least one engine component via the substantially consistent interface.

18. A distributed computer system, comprising:

a server configured to include

at least one engine having an engine interface, and providing one or more features to be executed;

at least one engine component configured to execute the one or more features of the engine by mapping a substantially consistent interface to the engine interface of the engine; and

a client configured to be connectable to said server and optionally configured to be connectable to another server, said client including an object manager layer communicable with and managing the at least one engine component stored on said server via the substantially consistent interface.

19. In a computer architecture including an engine management layer interfacing with a program specific application programmer interface (API) and providing engine management and administration, an engine configuration layer transforming API calls received from the program specific API into standardized calls, and an engine layer managing the standardized calls for each engine, said engine management layer configuring the computer architecture to perform at least one of the following computer implemented or computer assisted operations:

(a) loading and unloading engine dynamic link libraries into and out of memory for each engine;

(b) mapping at least one engine function to at least one corresponding engine object;

(c) providing general error detection and error correction for each engine;

(d) determining and matching arguments and returning values for mapping the at least one engine function to the at least one corresponding engine object; and

(e) managing error feedback from the at least one program specific API.

20. A computer implemented method utilizing a substantially consistent interface for individual object components that represent diverse technologies, comprising the steps of:

(a) migrating a plurality of engines to the consistent interface; and

(b) at least one of substantially automatically and substantially uniformly, managing the individual object components using a predefined object manager and the consistent interface.

21. A computer implemented process, comprising one or more of the following steps:

(a) providing the user an engine management function furnishing a protective wrapper for each function call associated with the engine, trapping errors, and providing error management and administration to prevent conditions associated with improper engine functioning;

(b) providing the user an engine configuration function transforming application programmer interface (API) calls received from the program specific API into standardized calls, the engine configuration function providing additional functionality, including safely loading and unloading the engine; and

(c) providing the user an engine function managing the standardized calls for each engine, thereby providing substantially uniform access to the engine and the engine settings associated with the engine.

22. A distributed computer implemented process including an object providing substantially uniform access to at least one engine having engine settings associated with the engine and an engine interface for accessing the engine settings and one or more features to be executed, comprising the steps of:

(a) executing, on at least one of the server and another server connectable to the server, at least one engine component to thereby execute the one or more features of the engine via a substantially consistent interface to the engine interface of the engine; and

(b) managing, on a client configured to be connectable to the server and optionally configured to be connectable to the another server, an object manager layer to manage the at least one engine component via the substantially consistent interface.

* * * * *

US006581088B1

(12) **United States Patent**
Jacobs et al.

(10) **Patent No.: US 6,581,088 B1**
(45) **Date of Patent: Jun. 17, 2003**

(54) **SMART STUB OR ENTERPRISE JAVA™ BEAN IN A DISTRIBUTED PROCESSING SYSTEM**

(75) Inventors: **Dean B. Jacobs**, Berkeley, CA (US); **Eric M. Halpern**, San Francisco, CA (US)

(73) Assignee: **Beas Systems, Inc.**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/405,260**

(22) Filed: **Sep. 23, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/107,167, filed on Nov. 5, 1998.

(51) **Int. Cl.**[7] **G06F 9/00**

(52) **U.S. Cl.** **709/105**; 709/100; 709/203

(58) **Field of Search** 709/100, 102, 709/103, 104, 107, 108, 310, 315, 316, 317, 328, 318, 319

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,465,365	A	11/1995	Winterbottom	707/104
5,475,819	A	12/1995	Miller et al.	395/200.03
5,564,070	A	10/1996	Want et al.	455/53.1
5,623,666	A	4/1997	Pike et al.	707/200
5,692,180	A	11/1997	Lee	707/10
5,701,484	A	12/1997	Artsy	709/242
5,701,502	A	12/1997	Baker et al.	709/201
5,764,982	A	6/1998	Madduri	707/10
5,774,660	A *	6/1998	Brendel et al.	395/200.31
5,790,548	A	8/1998	Sistanizadeh et al.	707/10
5,794,006	A	8/1998	Sanderman	707/10
5,805,804	A	9/1998	Laursen et al.	395/200.02
5,819,019	A	10/1998	Nelson	707/10
5,819,044	A	10/1998	Kawabe et al.	395/200.56
5,842,219	A	11/1998	High, Jr. et al.	707/103
5,850,449	A	12/1998	McManis	703/161
5,862,331	A	1/1999	Herriot	395/200.49
5,901,227	A	5/1999	Perlman	380/21
5,961,582	A	10/1999	Gaines	709/1
5,966,702	A	10/1999	Fresko et al.	707/1
5,974,441	A	10/1999	Rogers et al.	709/200
5,983,233	A	11/1999	Potonniee	707/103
5,983,351	A	11/1999	Glogau	713/201
5,999,988	A	12/1999	Pelegri-Llopart et al.	709/330
6,003,065	A	12/1999	Yan et al.	709/201
6,006,264	A *	12/1999	Colby et al.	709/226
6,016,505	A	1/2000	Badovinatz et al.	709/205
6,144,944	A *	11/2000	Kurtzman, II et al.	705/14

OTHER PUBLICATIONS

Brewing Distributed Applications With Java ORB Tools, Dec. 1996.*

Sun Microsystems, JavaBeans, Graham Hamilton, Jul. 1997.*

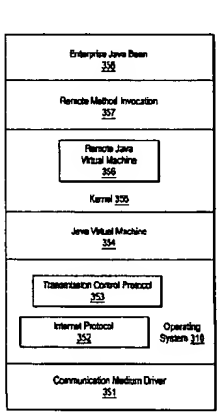
* cited by examiner

Primary Examiner—Majid Banankhah
(74) *Attorney, Agent, or Firm*—Fliesler, Dubb, Meyer & Lovejoy LLP

(57) **ABSTRACT**

A clustered enterprise Java™ distributed processing system is provided. The distributed processing system includes a first and a second computer coupled to a communication medium. The first computer includes a Java™ virtual machine (JVM) and kernel software layer for transferring messages, including a remote Java™ virtual machine (RJVM). The second computer includes a JVM and a kernel software layer having a RJVM. Messages are passed from a RJVM to the JVM in one computer to the JVM and RJVM in the second computer. Messages may be forwarded through an intermediate server or rerouted after a network reconfiguration. Each computer includes a Smart stub having a replica handler, including a load balancing software component and a failover software component. Each computer includes a duplicated service naming tree for storing a pool of Smart stubs at a node. The computers may be programmed in a stateless, stateless factory, or a stateful programming model. The clustered enterprise Java™ distributed processing system allows for enhanced scalability and fault tolerance.

74 Claims, 16 Drawing Sheets



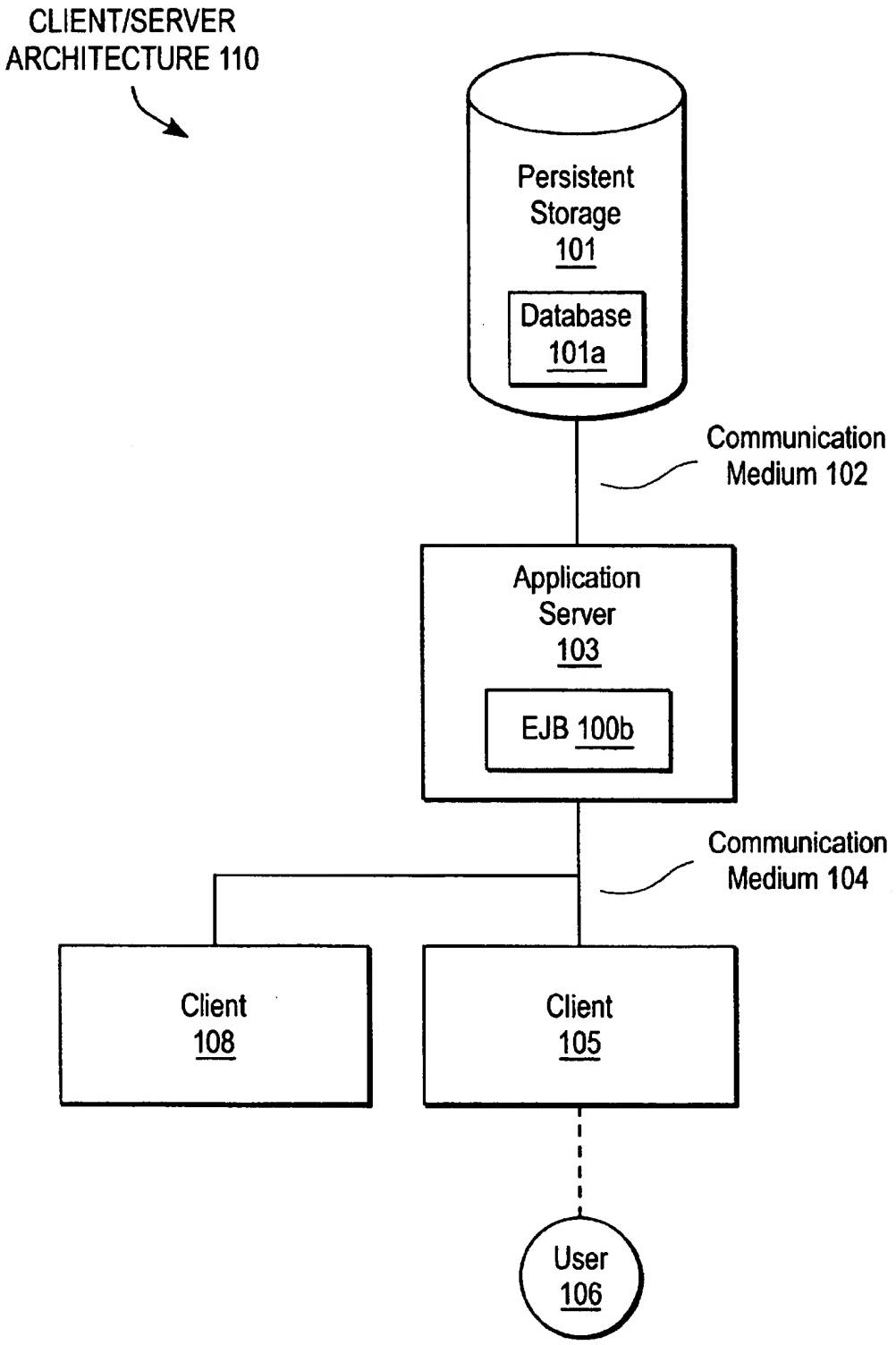


FIG. 1a (PRIOR ART)

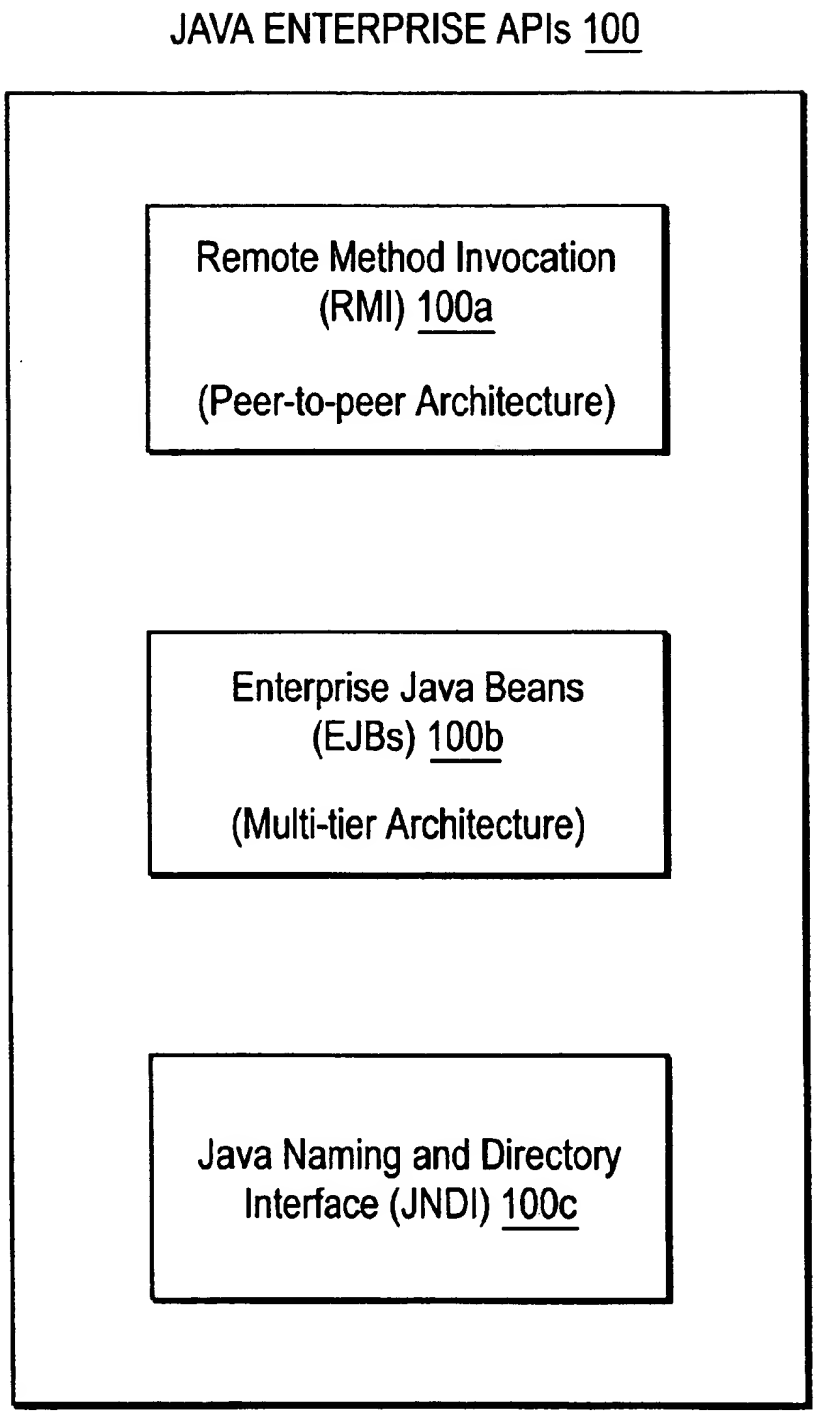


FIG. 1b (PRIOR ART)

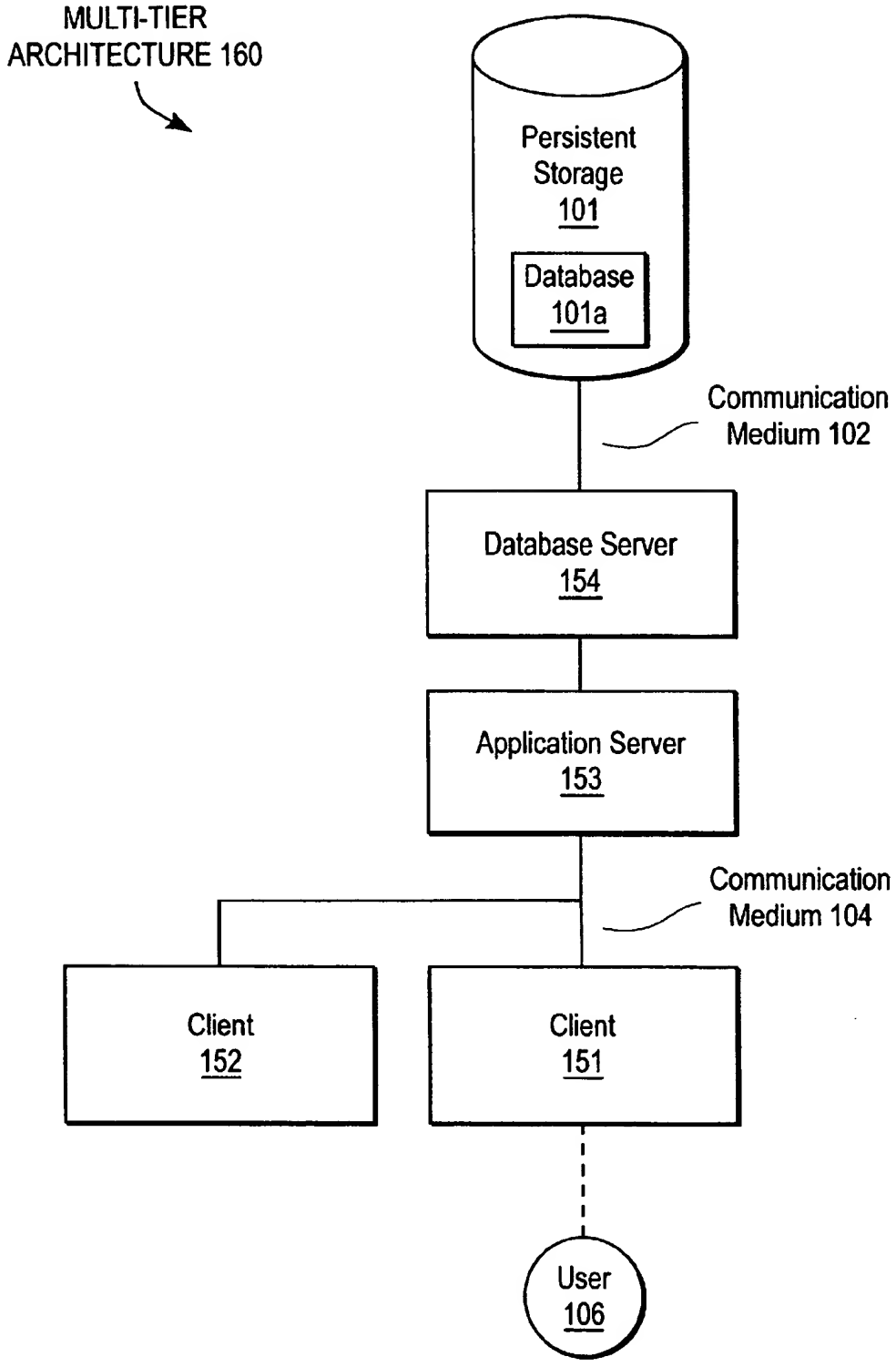


FIG. 1c (PRIOR ART)

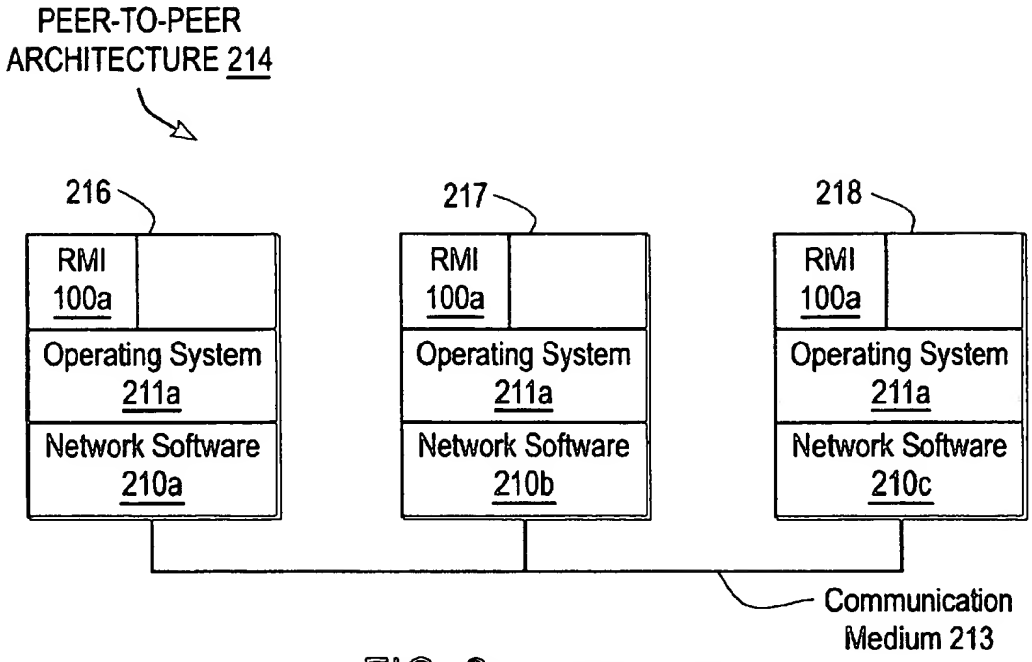


FIG. 2a (PRIOR ART)

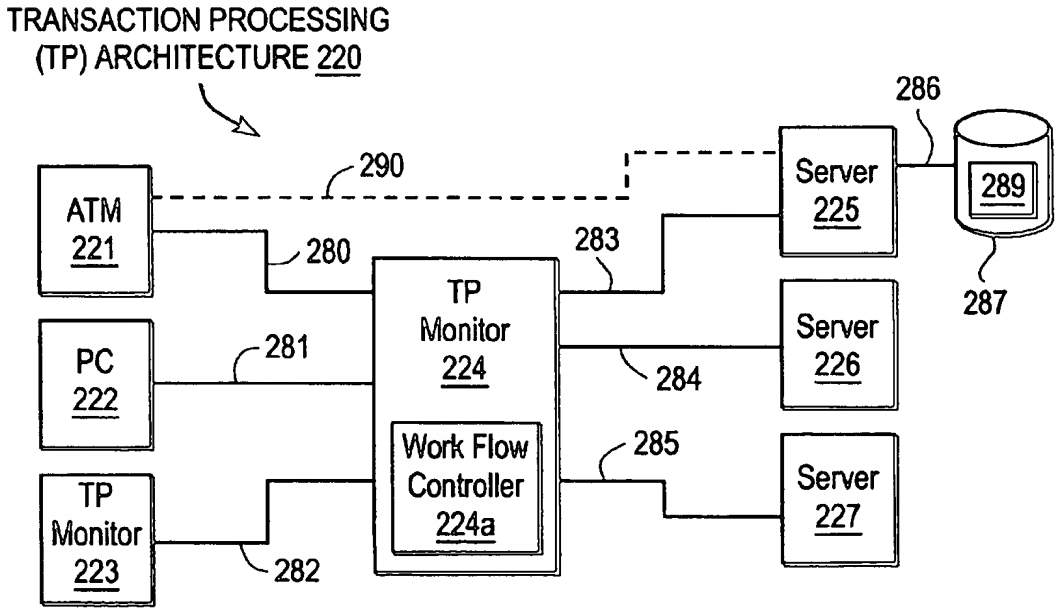


FIG. 2b (PRIOR ART)

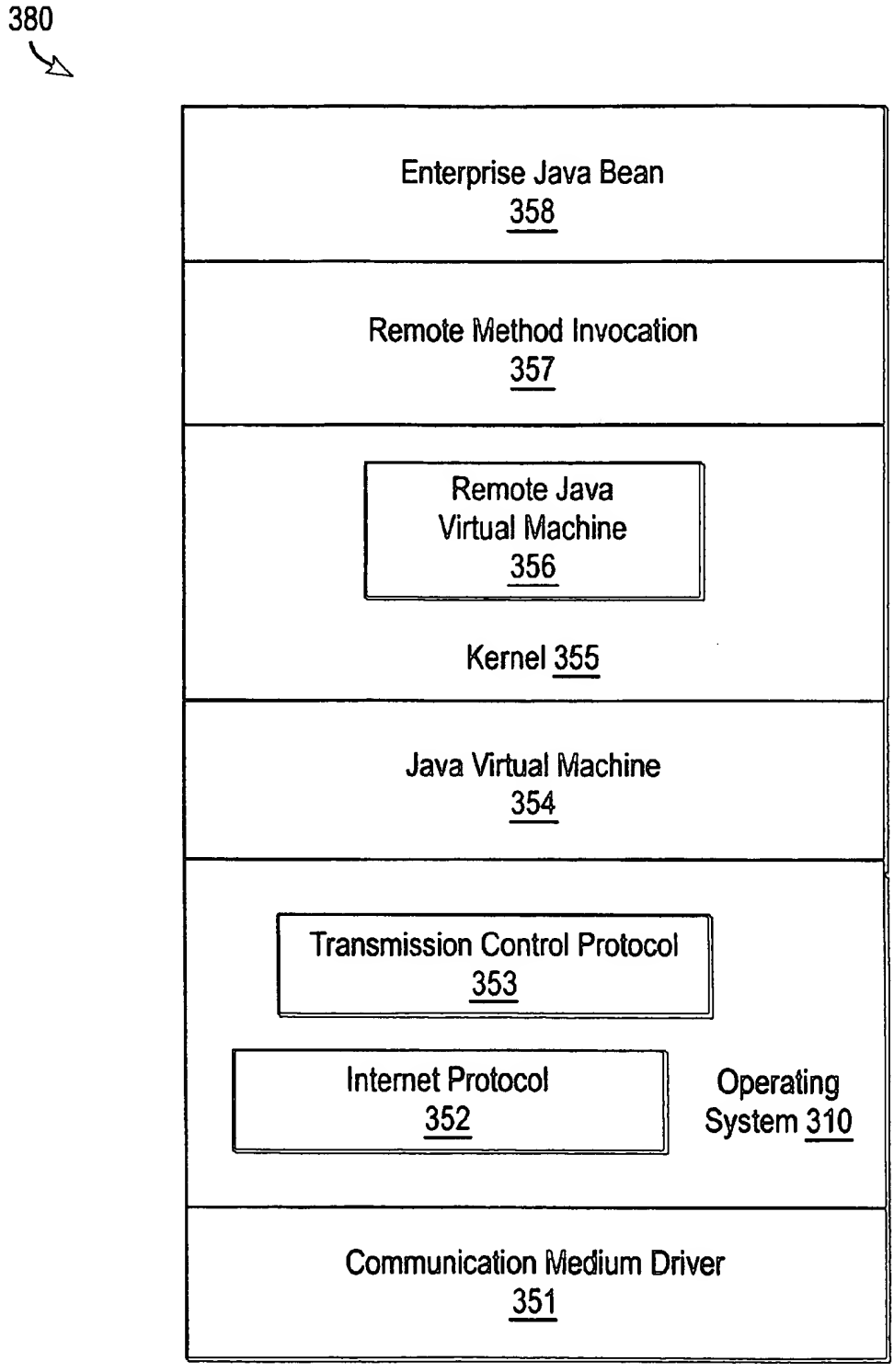

380
Enterprise Java Bean
358
Remote Method Invocation
357
Remote Java
Virtual Machine
356
Kernel 355
Java Virtual Machine
354
Transmission Control Protocol
353
Internet Protocol
352
Operating
System 310
Communication Medium Driver
351

FIG. 3a

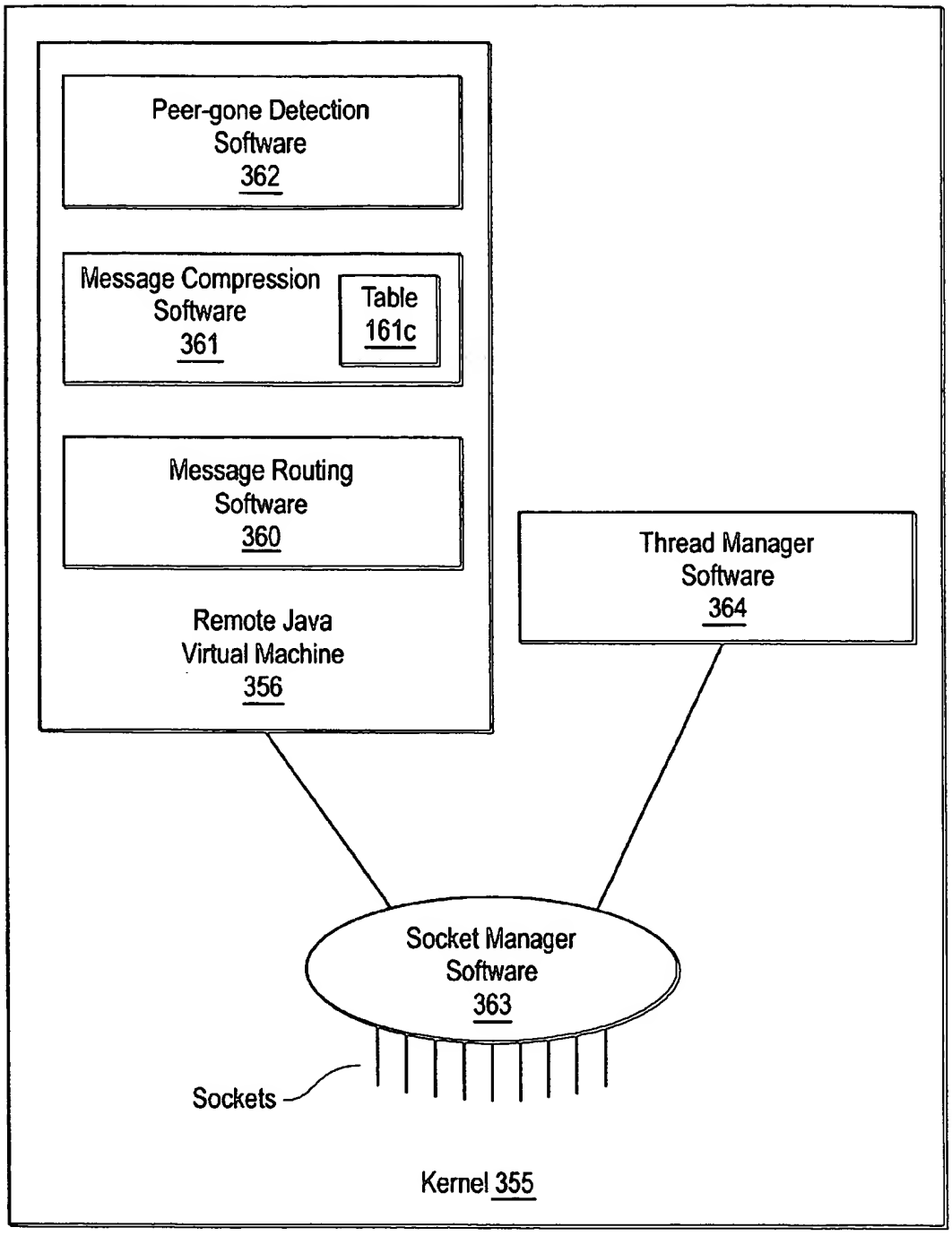

Peer-gone Detection Software 362
Message Compression Software 361
Table 161c
Message Routing Software 360
Remote Java Virtual Machine 356
Thread Manager Software 364
Socket Manager Software 363
Sockets
Kernel 355

FIG. 3b

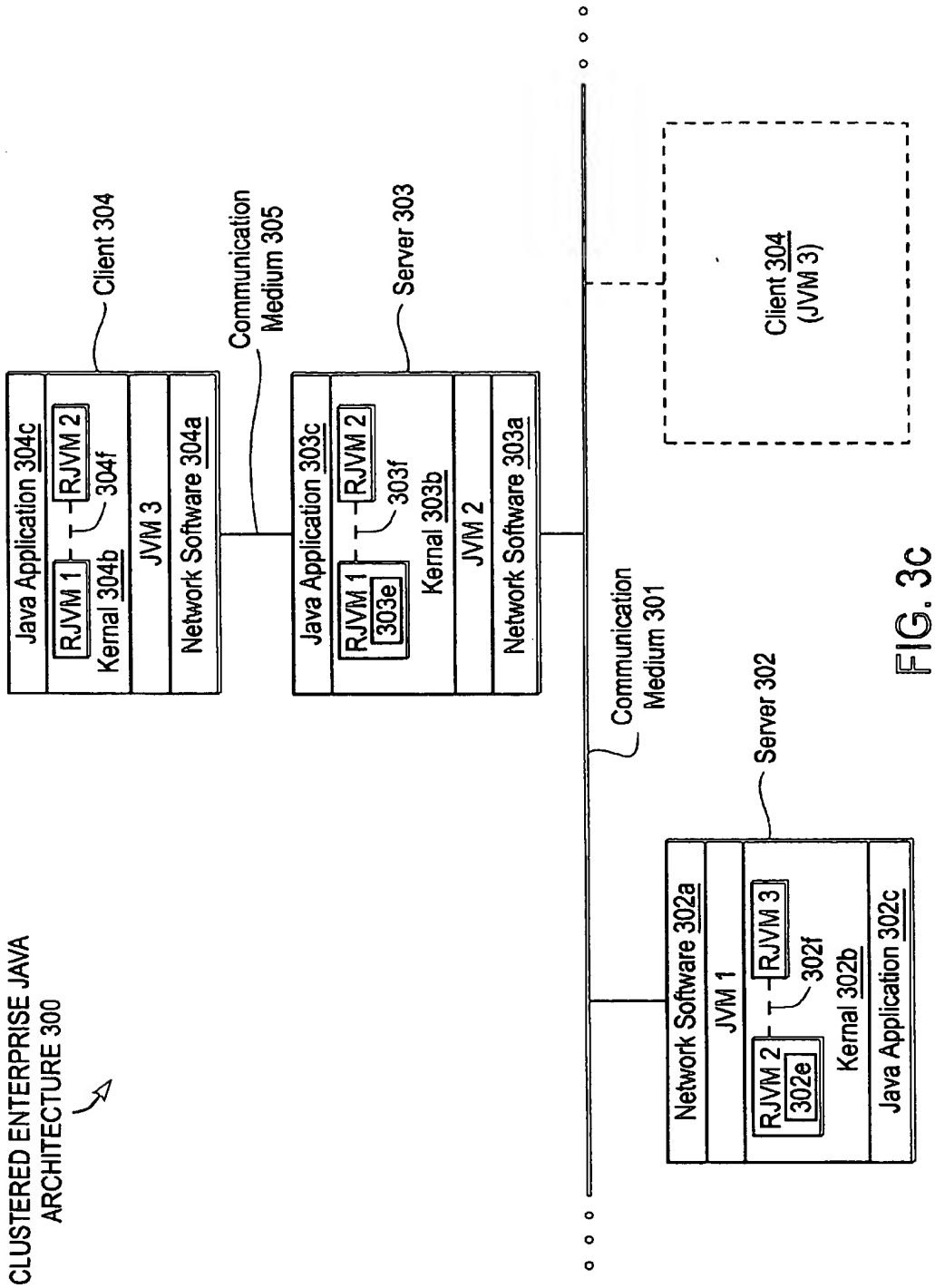

CLUSTERED ENTERPRISE JAVA ARCHITECTURE 300
Client 304
Java Application 304c
RJVM 1
RJVM 2
304f
Kernal 304b
JVM 3
Network Software 304a
Communication Medium 305
Server 303
Java Application 303c
RJVM 1
303e
RJVM 2
303f
Kernal 303b
JVM 2
Network Software 303a
Communication Medium 301
Client 304 (JVM 3)
Server 302
Network Software 302a
JVM 1
RJVM 2
302e
RJVM 3
302f
Kernal 302b
Java Application 302c

FIG. 3c

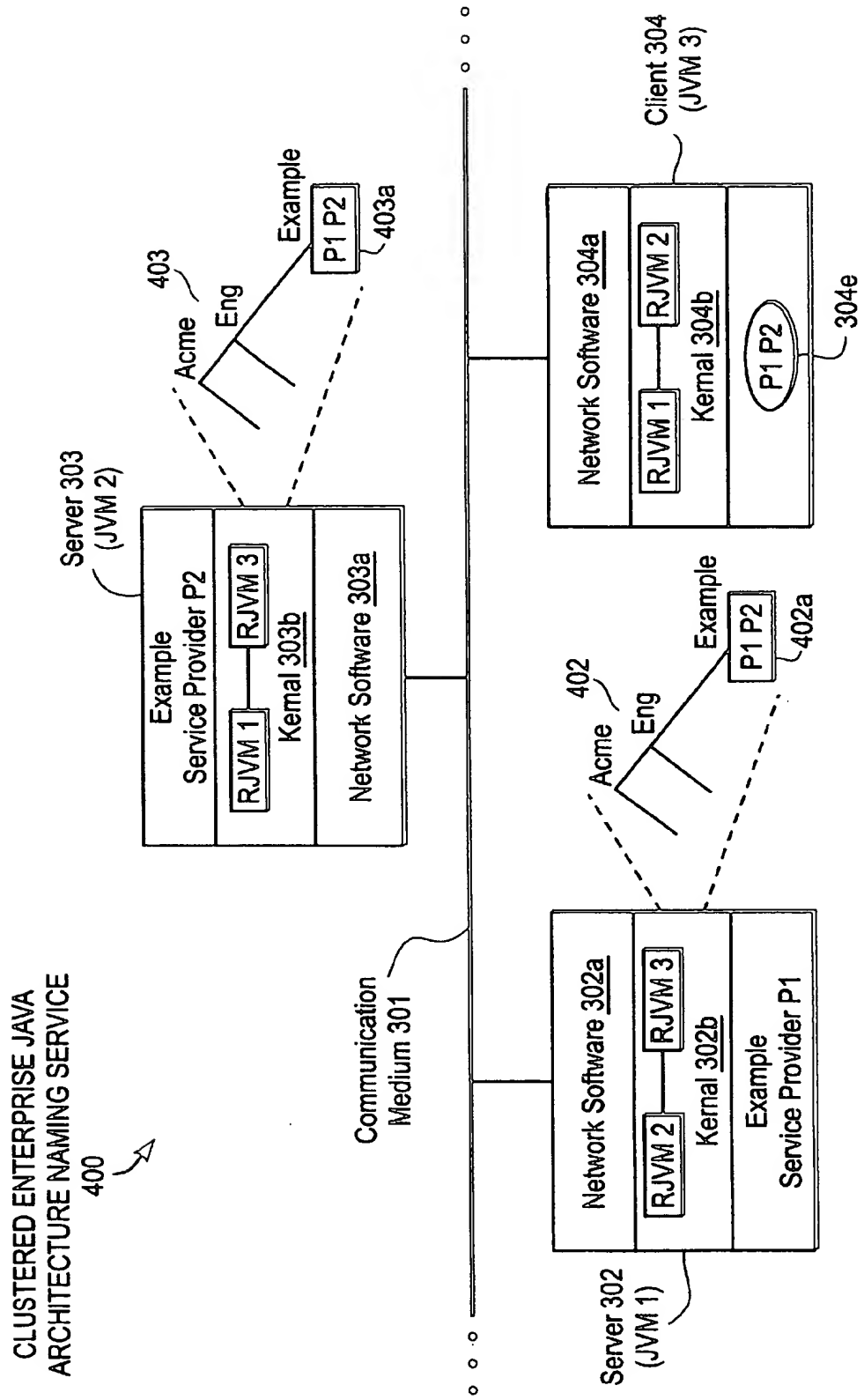

CLUSTERED ENTERPRISE JAVA ARCHITECTURE NAMING SERVICE
400
Communication Medium 301
Server 303 (JVM 2)
Example Service Provider P2
RJVM 1
RJVM 3
Kernal 303b
Network Software 303a
403
Acme
Eng
Example
P1 P2
403a
Server 302 (JVM 1)
Network Software 302a
RJVM 2
RJVM 3
Kernal 302b
Example Service Provider P1
402
Acme
Eng
Example
P1 P2
402a
Client 304 (JVM 3)
Network Software 304a
RJVM 1
RJVM 2
Kernal 304b
P1 P2
304e

FIG. 4

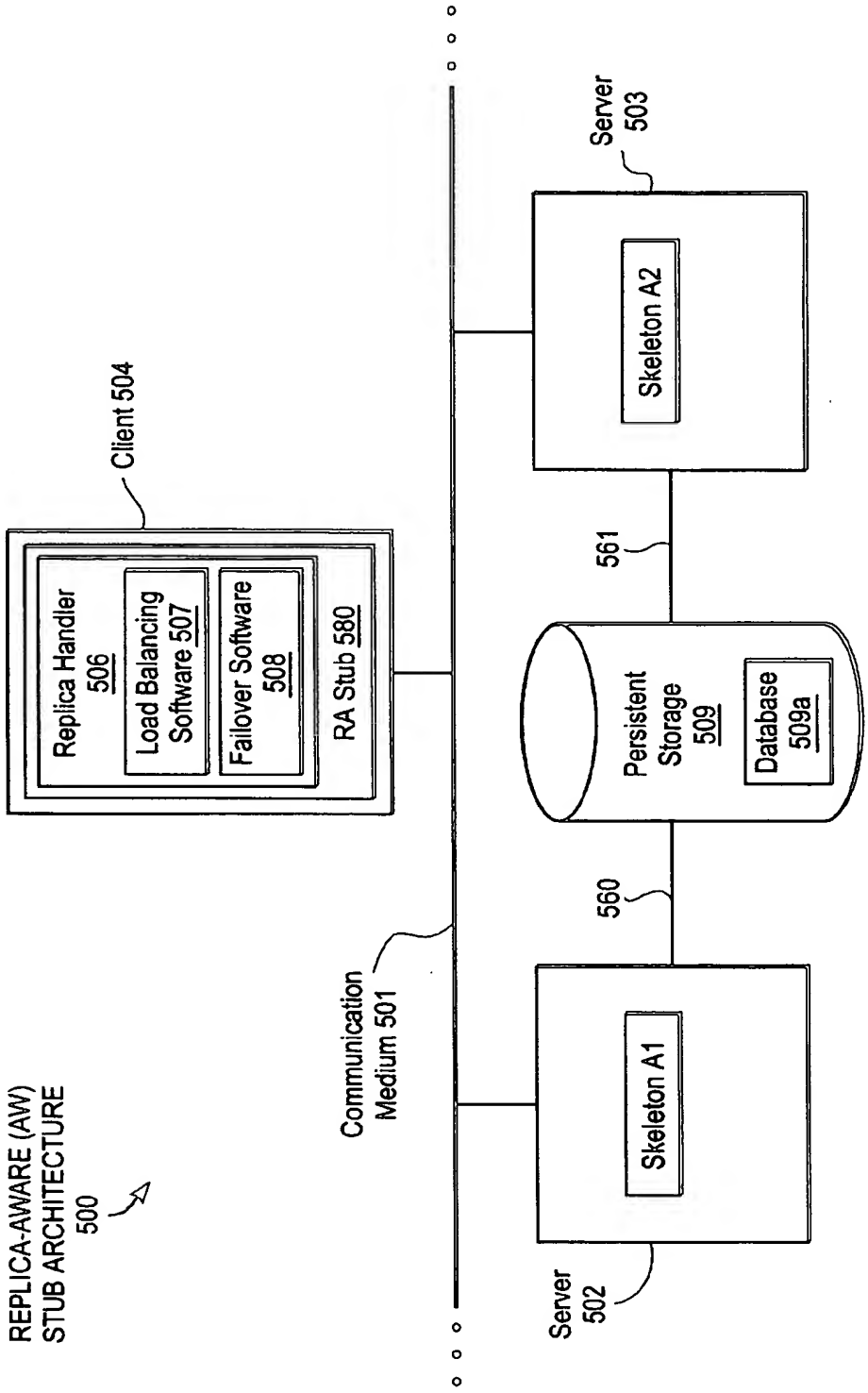

REPLICA-AWARE (AW)
STUB ARCHITECTURE
500
Client 504
Replica Handler
506
Load Balancing
Software 507
Failover Software
508
RA Stub 580
Communication
Medium 501
Server
502
Skeleton A1
560
Persistent
Storage
509
Database
509a
561
Server
503
Skeleton A2

FIG. 5a

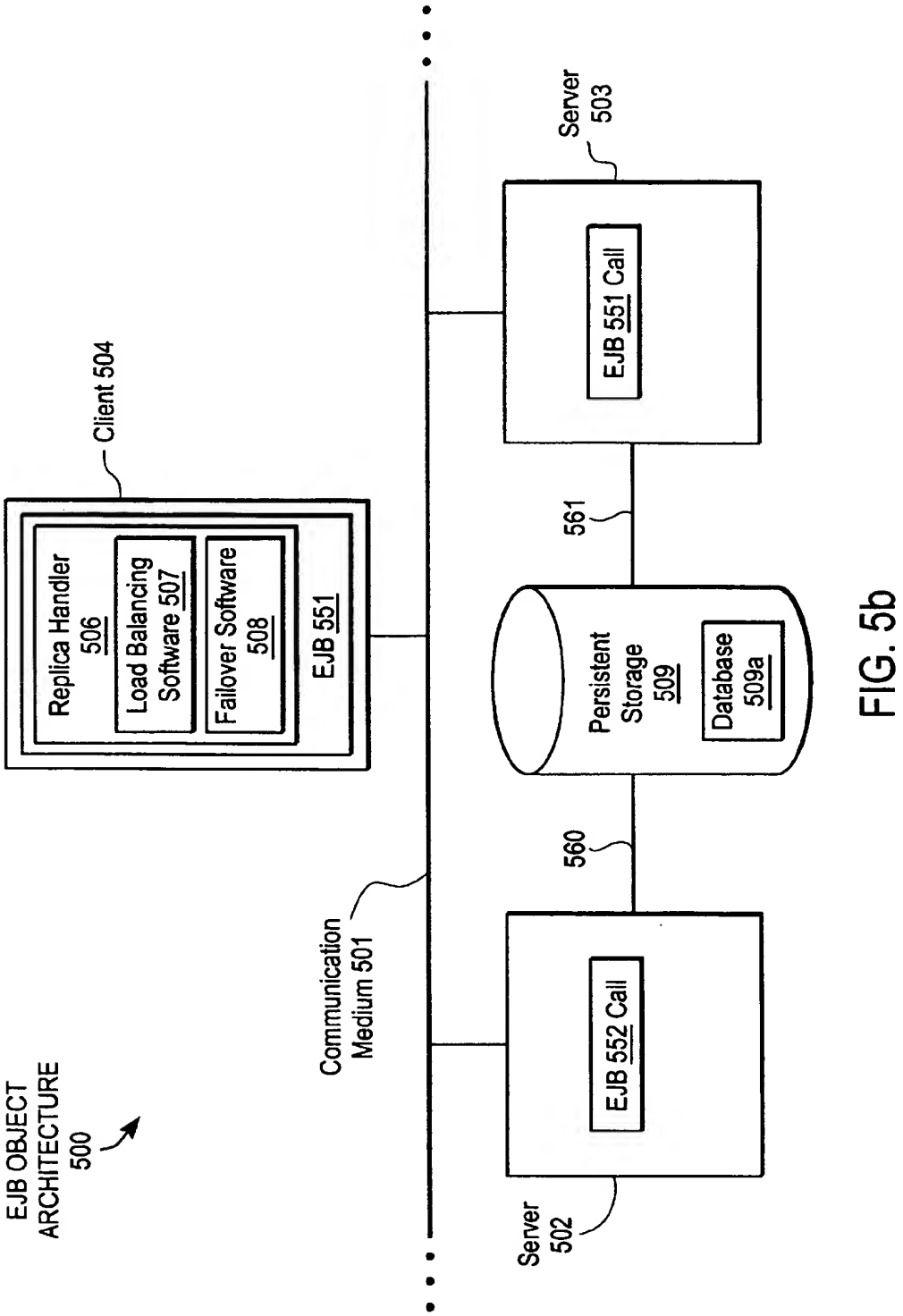

EJB OBJECT ARCHITECTURE 500
Client 504
Replica Handler 506
Load Balancing Software 507
Failover Software 508
EJB 551
Communication Medium 501
Server 503
EJB 551 Call
561
Persistent Storage 509
Database 509a
560
EJB 552 Call
Server 502

FIG. 5b

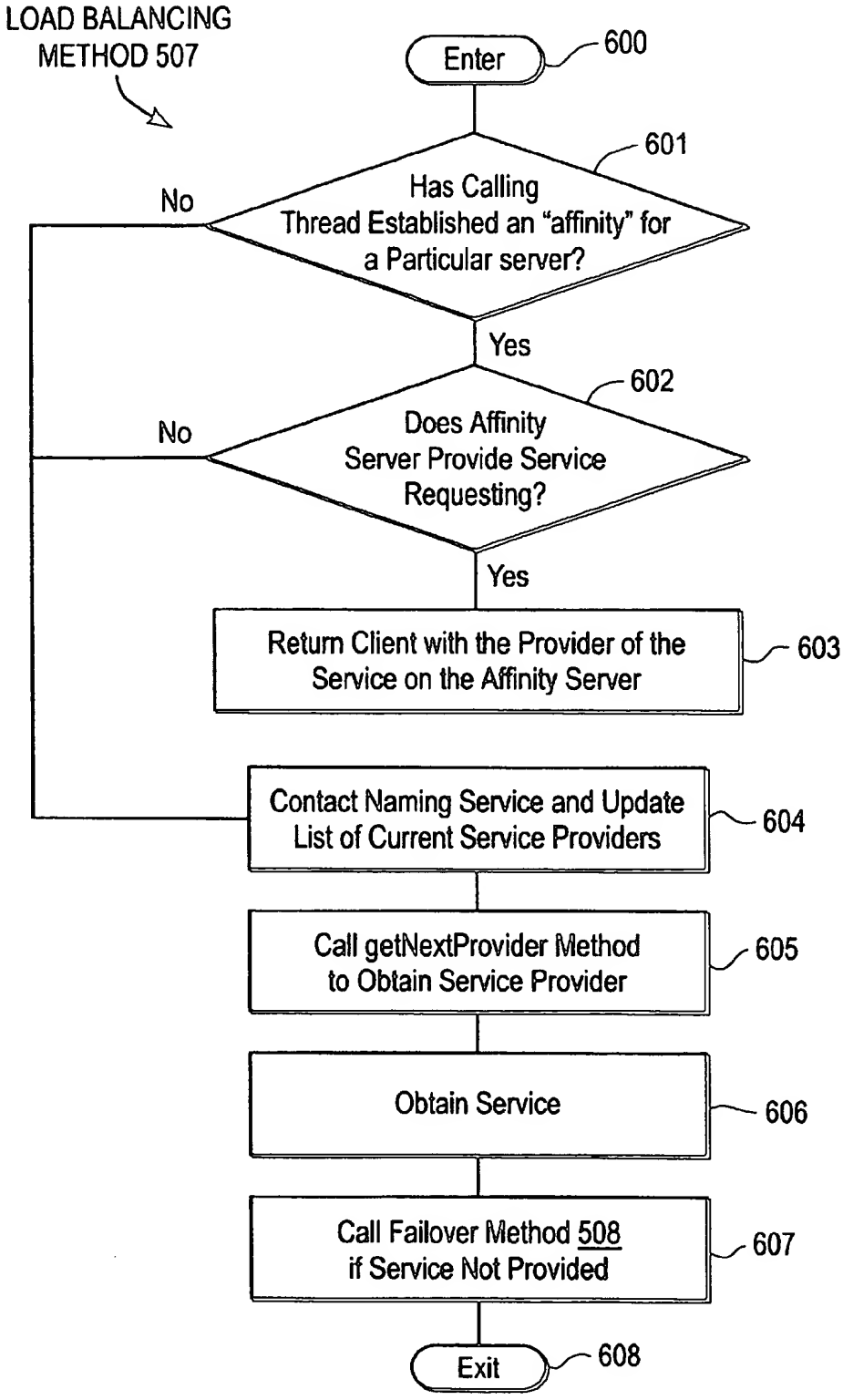

LOAD BALANCING METHOD 507
Enter
600
601
Has Calling Thread Established an "affinity" for a Particular server?
No
Yes
602
Does Affinity Server Provide Service Requesting?
No
Yes
Return Client with the Provider of the Service on the Affinity Server
603
Contact Naming Service and Update List of Current Service Providers
604
Call getNextProvider Method to Obtain Service Provider
605
Obtain Service
606
Call Failover Method 508 if Service Not Provided
607
Exit
608

FIG. 6a

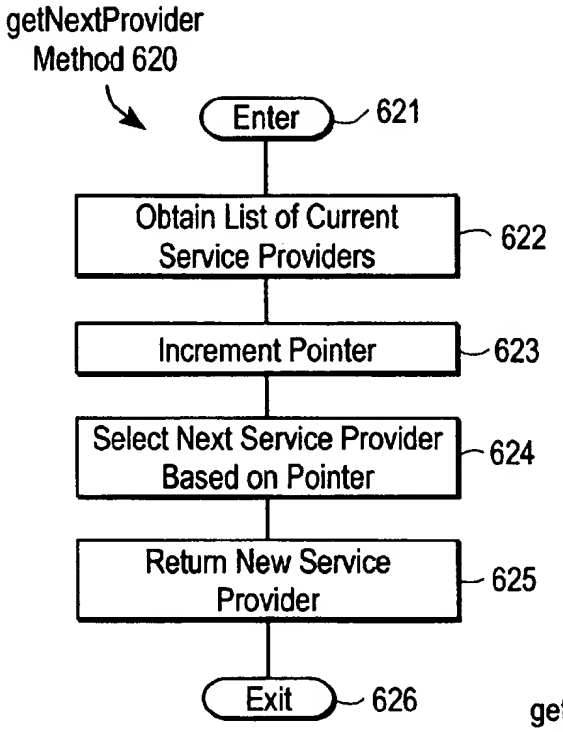

getNextProvider
Method 620
Enter
621
Obtain List of Current Service Providers
622
Increment Pointer
623
Select Next Service Provider Based on Pointer
624
Return New Service Provider
625
Exit
626

FIG. 6b

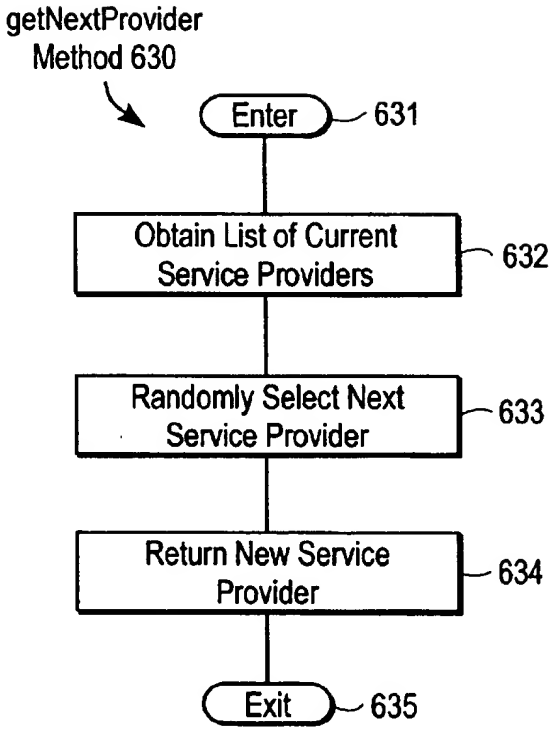

getNextProvider
Method 630
Enter
631
Obtain List of Current Service Providers
632
Randomly Select Next Service Provider
633
Return New Service Provider
634
Exit
635

FIG. 6c

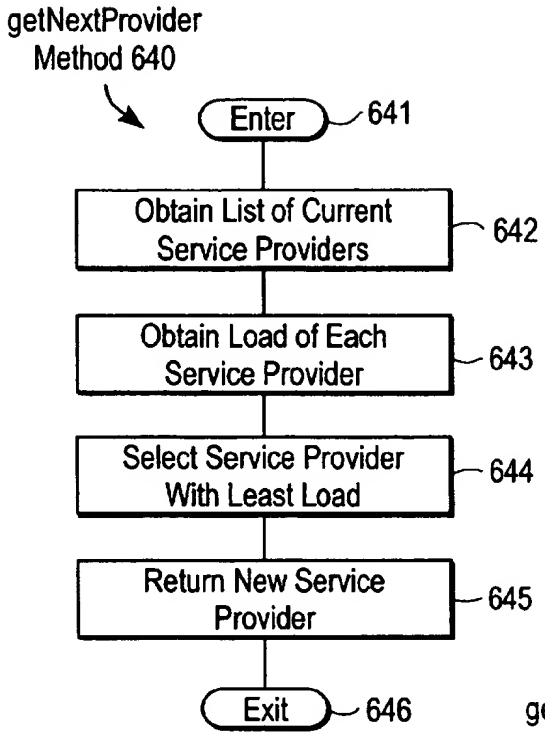

getNextProvider
Method 640
Enter
641
Obtain List of Current Service Providers
642
Obtain Load of Each Service Provider
643
Select Service Provider With Least Load
644
Return New Service Provider
645
Exit
646

FIG. 6d

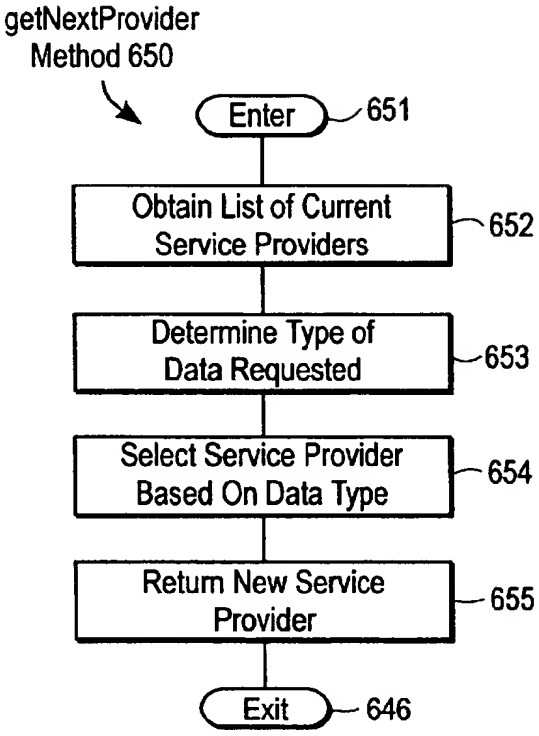

getNextProvider
Method 650
Enter
651
Obtain List of Current Service Providers
652
Determine Type of Data Requested
653
Select Service Provider Based On Data Type
654
Return New Service Provider
655
Exit
646

FIG. 6e

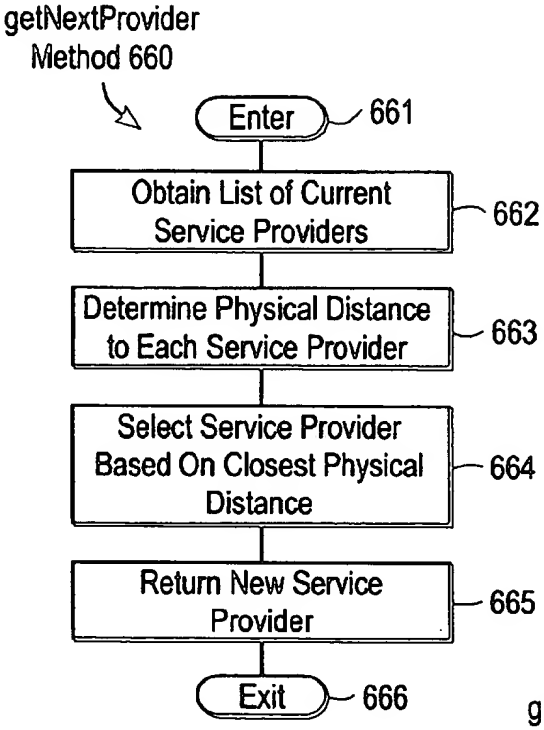

getNextProvider
Method 660
Enter
661
Obtain List of Current Service Providers
662
Determine Physical Distance to Each Service Provider
663
Select Service Provider Based On Closest Physical Distance
664
Return New Service Provider
665
Exit
666

FIG. 6f

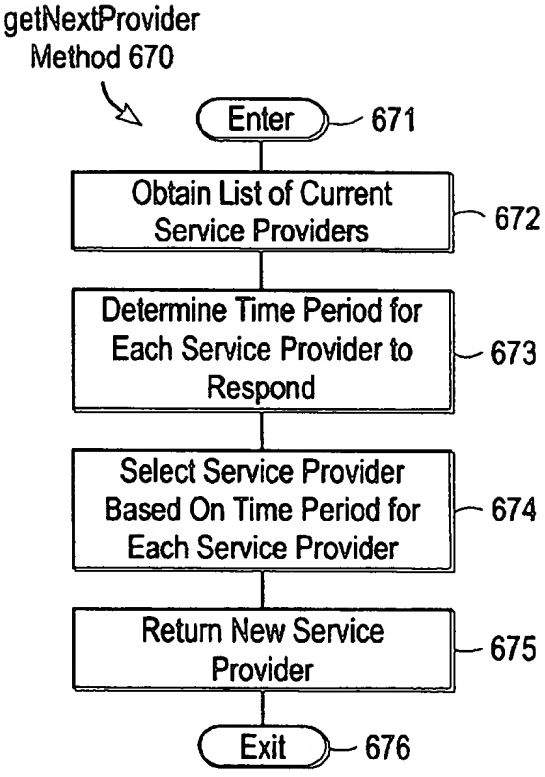

getNextProvider
Method 670
Enter
671
Obtain List of Current Service Providers
672
Determine Time Period for Each Service Provider to Respond
673
Select Service Provider Based On Time Period for Each Service Provider
674
Return New Service Provider
675
Exit
676

FIG. 6g

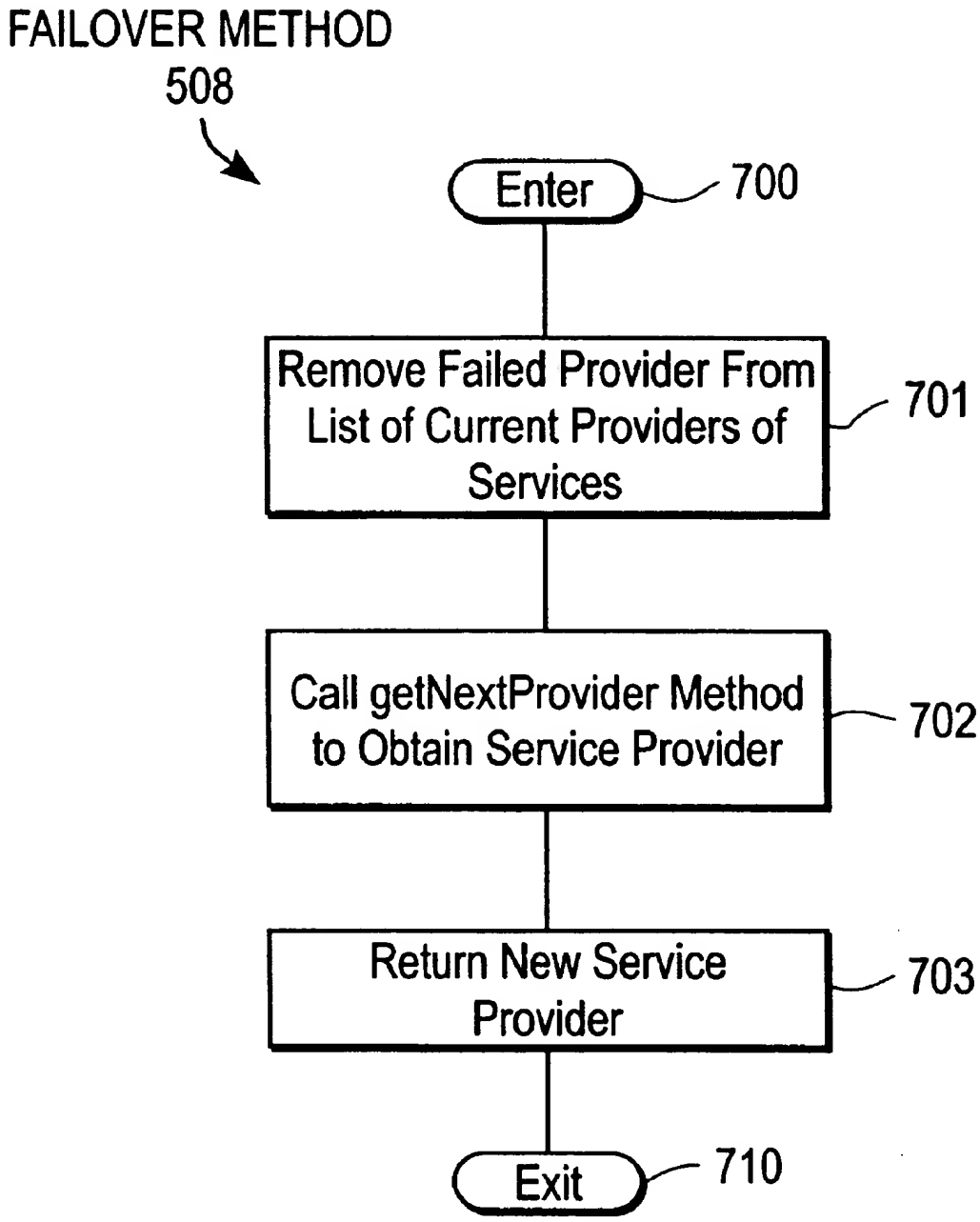

FAILOVER METHOD
508
Enter
700
Remove Failed Provider From List of Current Providers of Services
701
Call getNextProvider Method to Obtain Service Provider
702
Return New Service Provider
703
Exit
710

FIG. 7

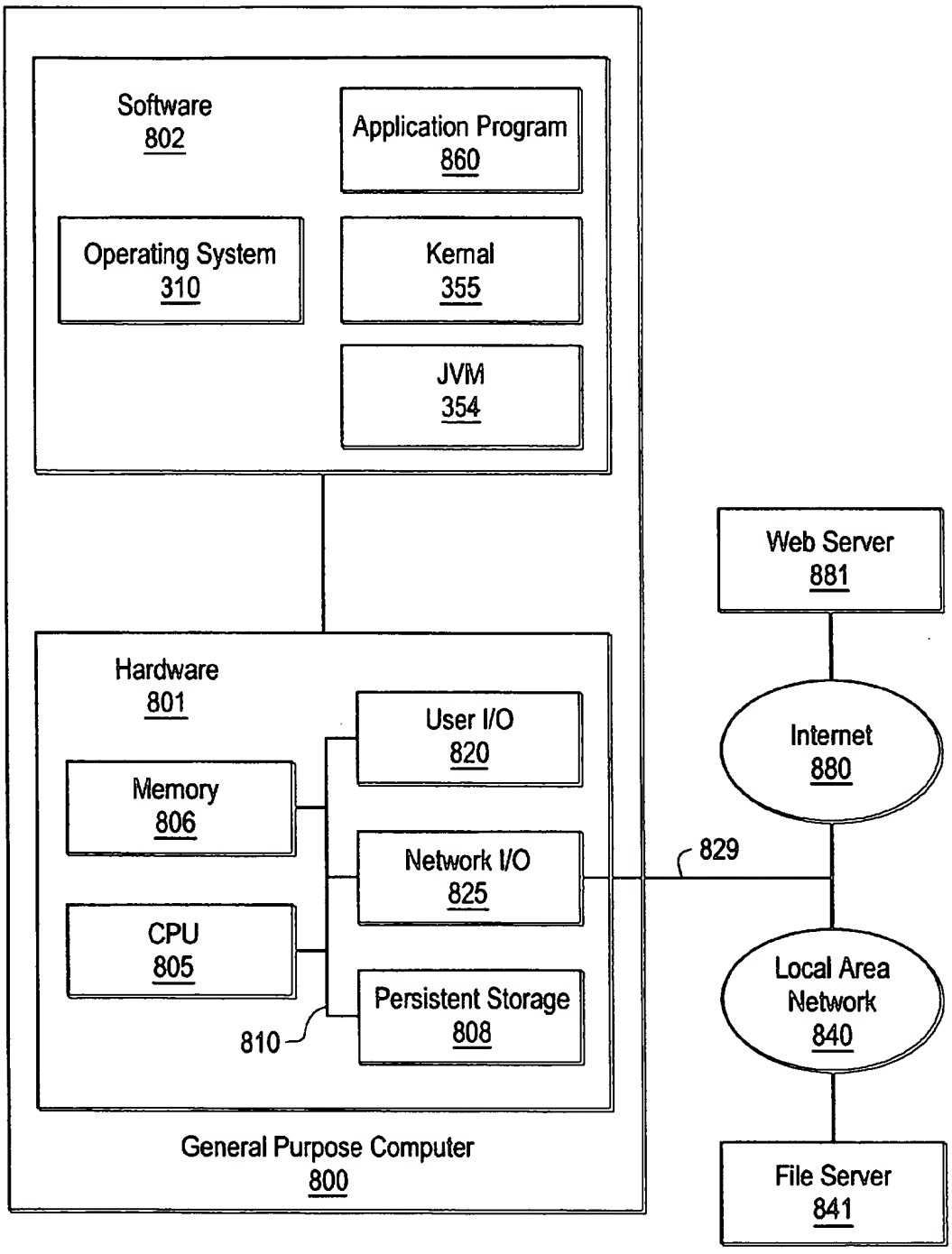

Software
802
Application Program
860
Operating System
310
Kernal
355
JVM
354
Web Server
881
Hardware
801
User I/O
820
Internet
880
Memory
806
Network I/O
825
829
CPU
805
Local Area
Network
840
Persistent Storage
808
810
General Purpose Computer
800
File Server
841

FIG. 8

SMART STUB OR ENTERPRISE JAVA™ BEAN IN A DISTRIBUTED PROCESSING SYSTEM

CROSS REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Application No. 60/107,167, filed Nov. 5, 1998.

The following copending U.S. patent applications are assigned to the assignee of the present application, and their disclosures are incorporated herein by reference:

(A) Ser. No. 09/405,318 filed Sep. 23, 1999 by Dean B. Jacobs and Anno R. Langen, and originally entitled, "CLUSTERED ENTERPRISE JAVA™ HAVING A MESSAGE PASSING KERNEL IN A DISTRIBUTED PROCESSING SYSTEM";

(B) Ser. No. 09/405,508 filed Sep. 23, 1999 by Dean B. Jacobs and Eric M. Halpern, and entitled, "A DUPLICATED NAMING SERVICE IN A DISTRIBUTED PROCESSING SYSTEM"; and

(C) Ser. No. 09/405,500 filed Sep. 23, 1999 by Dean B. Jacobs and Anno R. Langen, and originally entitled, "CLUSTERED ENTERPRISE JAVA™ IN A SECURE DISTRIBUTED PROCESSING SYSTEM".

FIELD OF THE INVENTION

The present invention relates to distributed processing systems and, in particular, computer software in distributed processing systems.

BACKGROUND OF THE INVENTION

There are several types of distributed processing systems. Generally, a distributed processing system includes a plurality of processing devices, such as two computers coupled to a communication medium. Communication mediums may include wired mediums, wireless mediums, or combinations thereof, such as an Ethernet local area network or a cellular network. In a distributed processing system, at least one processing device may transfer information on the communication medium to another processing device.

Client/server architecture **110** illustrated in FIG. **1***a* is one type of distributed processing system. Client/server architecture **110** includes at least two processing devices, illustrated as client **105** and application server **103**. Additional clients may also be coupled to communication medium **104**, such as client **108**.

Typically, server **103** hosts business logic and/or coordinates transactions in providing a service to another processing device, such as client **103** and/or client **108**. Application server **103** is typically programmed with software for providing a service. The software may be programmed using a variety of programming models, such as Enterprise Java™ Bean ("EJB") **100***b* as illustrated in FIGS. **1***a–b*. The service may include, for example, retrieving and transferring data from a database, providing an image and/or calculating an equation. For example, server **103** may retrieve data from database **101***a* in persistent storage **101** over communication medium **102** in response to a request from client **105**. Application server **103** then may transfer the requested data over communication medium **104** to client **105**.

A client is a processing device which utilizes a service from a server and may request the service. Often a user **106** interacts with client **106** and may cause client **105** to request service over a communication medium **104** from application server **103**. A client often handles direct interactions with end users, such as accepting requests and displaying results.

A variety of different types of software may be used to program application server **103** and/or client **105**. One programming language is the Java™ programming language. Java™ application object code is loaded into a Java™ virtual machine ("JVM"). A JVM is a program loaded onto a processing device which emulates a particular machine or processing device. More information on the Java™ programming language may be obtained at http://www.javasoft.com, which is incorporated by reference herein.

FIG. **1***b* illustrates several Java™ Enterprise Application Programming Interfaces ("APIs") **100** that allow Java™ application code to remain independent from underlying transaction systems, data-bases and network infrastructure. Java™ Enterprise APIs **100** include, for example, remote method invocation ("RMI") **100***a*, EJBs **100***b*, and Java™ Naming and Directory Interface (JNDI) **100***c*.

RMI **100***a* is a distributed programming model often used in peer-to-peer architecture described below. In particular, a set of classes and interfaces enables one Java™ object to call the public method of another Java™ object running on a different JVM.

An instance of EJB **100***b* is typically used in a client/server architecture described above. An instance of EJB **100***b* is a software component or a reusable pre-built piece of encapsulated application code that can be combined with other components. Typically, an instance of EJB **100***b* contains business logic. An EJB **100***b* instance stored on server **103** typically manages persistence, transactions, concurrency, threading, and security.

JNDI **100***c* provides directory and naming functions to Java™ software applications.

Client/server architecture **110** has many disadvantages. First, architecture **110** does not scale well because server **103** has to handle many connections. In other words, the number of clients which may be added to server **103** is limited. In addition, adding twice as many processing devices (clients) does not necessarily provide you with twice as much performance. Second, it is difficult to maintain application code on clients **105** and **108**. Third, architecture **110** is susceptible to system failures or a single point of failure. If server **101** fails and a backup is not available, client **105** will not be able to obtain the service.

FIG. **1***c* illustrates a multi-tier architecture **160**. Clients **151**, **152** manage direct interactions with end users, accepting requests and display results. Application server **153** hosts the application code, coordinates communications, synchronizations, and transactions. Database server **154** and portable storage device **155** provides durable transactional management of the data.

Multi-tier architecture **160** has similar client/server architecture **110** disadvantages described above.

FIG. **2** illustrates peer-to-peer architecture **214**. Processing devices **216**, **217** and **218** are coupled to communication medium **213**. Processing devices **216**, **217**, and **218** include network software **210***a*, **210***b*, and **210***c* for communicating over medium **213**. Typically, each processing device in a peer-to-peer architecture has similar processing capabilities and applications. Examples of peer-to-peer program models include Common Object Request Broker Architecture ("CORBA") and Distributed Object Component Model ("DCOM") architecture.

In a platform specific distributed processing system, each processing device may run the same operating system. This

allows the use of proprietary hardware, such as shared disks, multi-tailed disks, and high speed interconnects, for communicating between processing devices. Examples of platform-specific distributed processing systems include IBM® Corporation's S/390® Parallel Sysplex®, Compaq's Tandem Division Himalaya servers, Compaq's Digital Equipment Corporation™ (DEC™) Division OpenVMS™ Cluster software, and Microsoft® Corporation Windows NT® cluster services (Wolfpack).

FIG. **2***b* illustrates a transaction processing (TP) architecture **220**. In particular, TP architecture **220** illustrates a BEA® Systems, Inc. TUXEDO® architecture. TP monitor **224** is coupled to processing devices ATM **221**, PC **222**, and TP monitor **223** by communication medium **280**, **281**, and **282**, respectively. ATM **221** may be an automated teller machine, PC **222** may be a personal computer, and TP monitor **223** may be another transaction processor monitor. TP monitor **224** is coupled to back-end servers **225**, **226**, and **227** by communication mediums **283**, **284**, and **285**. Server **225** is coupled to persistent storage device **287**, storing database **289**, by communication medium **286**. TP monitor **224** includes a workflow controller **224***a* for routing service requests from processing devices, such as ATM **221**, PC **222**, or TP monitor **223**, to various servers such as server **225**, **226** and **227**. Work flow controller **224***a* enables (1) workload balancing between servers, (2) limited scalability or allowing for additional servers and/or clients, (3) fault tolerance of redundant backend servers (or a service request may be sent by a workflow controller to a server which has not failed), and (4) session concentration to limit the number of simultaneous connections to back-end servers. Examples of other transaction processing architectures include IBM® Corporation's CICS®, Compaq's Tandem Division Pathway/Ford/TS, Compaq's DEC™ ACMS, and Transarc Corporation's Encina.

TP architecture **220** also has many disadvantages. First, a failure of a single processing device or TP monitor **224** may render the network inoperable. Second, the scalability or number of processing devices (both servers and clients) coupled to TP monitor **224** may be limited by TP monitor **224** hardware and software. Third, flexibility in routing a client request to a server is limited. For example, if communication medium **280** is inoperable, but communication medium **290** becomes available, ATM **221** typically may not request service directly from server **225** over communication medium **290** and must access TP monitor **224**. Fourth, a client typically does not know the state of a back-end server or other processing device. Fifth, no industry standard software or APIs are used for load balancing. And sixth, a client typically may not select a particular server even if the client has relevant information which would enable efficient service.

Therefore, it is desirable to provide a distributed processing system and, in particular, distributed processing system software that has the advantages of the prior art distributed processing systems without the inherent disadvantages. The software should allow for industry standard APIs which are typically used in either client/server, multi-tier, or peer-to-peer distributed processing systems. The software should support a variety of computer programming models. Further, the software should enable (1) enhanced fault tolerance, (2) efficient scalability, (3) effective load balancing, and (4) session concentration control. The improved computer software should allow for rerouting or network reconfiguration. Also, the computer software should allow for the determination of the state of a processing device.

SUMMARY OF THE INVENTION

An improved distributed processing system is provided and, in particular, computer software for a distributed processing system is provided. The computer software improves the fault tolerance of the distributed processing system as well as enables efficient scalability. The computer software allows for efficient load balancing and session concentration. The computer software supports rerouting or reconfiguration of a computer network. The computer software supports a variety of computer programming models and allows for the use of industry standard APIs that are used in both client/server and peer-to-peer distributed processing architectures. The computer software enables a determination of the state of a server or other processing device. The computer software also supports message forwarding under a variety of circumstances, including a security model.

According to one aspect of the present invention, a distributed processing system comprises a communication medium coupled to a first processing device and a second processing device. The first processing device includes a first software program emulating a processing device ("JVM1") including a first kernel software layer having a data structure ("RJVM1"). The second processing device includes a first software program emulating a processing device ("JVM2") including a first kernel software layer having a data structure ("RJVM2"). A message from the first processing device is transferred to the second processing device through the first kernel software layer and the first software program in the first processing device to the first kernel software layer and the first software program in the second processing device.

According to another aspect of the present invention, the first software program in the first processing device is a Java™ virtual machine ("JVM") and the data structure in the first processing device is a remote Java™ virtual machine ("RJVM"). Similarly, the first software program in the second processing device is a JVM and the data structure in the second processing device is a RJVM. The RJVM in the second processing device corresponds to the JVM in the first processing device.

According to another aspect of the present invention, the RJVM in the first processing device includes a socket manager software component, a thread manager software component, a message routing software component, a message compression software component, and/or a peer-gone detection software component.

According to another aspect of the present invention, the first processing device communicates with the second processing device using a protocol selected from the group consisting of Transmission Control Protocol ("TCP"), Secure Sockets Layer ("SSL"), Hypertext Transport Protocol ("HTTP") tunneling, and Internet InterORB Protocol ("IIOP") tunneling.

According to another aspect of the present invention, the first processing device includes memory storage for a Java™ application.

According to another aspect of the present invention, the first processing device is a peer of the second processing device. Also, the first processing device is a server and the second processing device is a client.

According to another aspect of the present invention, a second communication medium is coupled to the second processing device. A third processing device is coupled to the second communication medium. The third processing device includes a first software program emulating a processing device ("JVM3"), including a kernel software layer having a first data structure ("RJVM1"), and a second data structure ("RJVM2").

According to still another aspect of the present invention, the first processing device includes a stub having a replica-

handler software component. The replica-handler software component includes a load balancing software component and a failover software component.

According to another aspect of the present invention, the first processing device includes an Enterprise Java™ Bean object.

According to still another aspect of the present invention, the first processing device includes a naming tree having a pool of stubs stored at a node of the tree and the second processing device includes a duplicate of the naming tree.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateless program model and the application program includes a stateless session bean.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateless factory program model and the application program includes a stateful session bean.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateful program model and the application program includes an entity session bean.

According to still another aspect of the present invention, an article of manufacture including an information storage medium is provided. The article of manufacture comprises a first set of digital information for transferring a message from a RJVM in a first processing device to a RJVM in a second processing device.

According to another aspect of the present invention, the article of manufacture comprises a first set of digital information, including a stub having a load balancing software program for selecting a service provider from a plurality of service providers.

According to another aspect of the present invention, the stub has a failover software component for removing a failed service provider from the plurality of service providers.

According to another aspect of the present invention, the load balancing software component selects a service provider based on an affinity for a particular service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider in a round robin manner.

According to another aspect of the present invention, the load balancing software component randomly selects a service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the load of each service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the data type requested.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the closest physical service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon a time period in which each service provider responds.

According to another aspect of the present invention, the article of manufacture comprises a first set of digital information, including an Enterprise Java™ Bean object for selecting a service provider from a plurality of service providers.

According to another aspect of the present invention, a stub is stored in a processing device in a distributed processing system. The stub includes a method comprising the steps of obtaining a list of service providers and selecting a service provider from the list of service providers.

According to another aspect of the present invention, the method further includes removing a failed service provider from the list of service providers.

According to still another aspect of the present invention, an apparatus comprises a communication medium coupled to a first processing device and a second processing device. The first processing device stores a naming tree including a remote method invocation ("RMI") stub for accessing a service provider. The second processing device has a duplicate naming tree and the service provider.

According to another aspect of the present invention, the naming tree has a node including a service pool of current service providers.

According to another aspect of the present invention, the service pool includes a stub.

According to another aspect of the present invention, a distributed processing system comprises a first computer coupled to a second computer. The first computer has a naming tree, including a remote invocation stub for accessing a service provider. The second computer has a replicated naming tree and the service provider.

According to another aspect of the present invention, a distributed processing system comprising a first processing device coupled to a second processing device is provided. The first processing device has a JVM and a first kernel software layer including a first RJVM. The second processing device includes a first JVM and a first kernel software layer including a second RJVM. A message may be transferred from the first processing device to the second processing device when there is not a socket available between the first JVM and the second JVM.

According to another aspect of the present invention, the first processing device is running under an applet security model, behind a firewall or is a client, and the second processing device is also a client.

Other aspects and advantages of the present invention can be seen upon review of the figures, the detailed description, and the claims which follow.

BRIEF DESCRIPTION OF THE FIGURES

FIG. 1*a* illustrates a prior art client/server architecture;

FIG. 1*b* illustrates a prior art Java™ enterprise APIs;

FIG. 1*c* illustrates a multi-tier architecture;

FIG. 2*a* illustrates a prior art peer-to-peer architecture;

FIG. 2*b* illustrates a prior art transaction processing architecture;

FIG. 3*a* illustrates a simplified software block diagram of an embodiment of the present invention;

FIG. 3*b* illustrates a simplified software block diagram of the kernel illustrated in FIG. 3*a;*

FIG. 3*c* illustrates a clustered enterprise Java™ architecture;

FIG. 4 illustrates a clustered enterprise Java™ naming service architecture;

FIG. 5*a* illustrates a smart stub architecture;

FIG. 5*b* illustrates an EJB object architecture;

FIG. 6*a* is a control flow chart illustrating a load balancing method;

FIGS. 6*b–g* are control flow charts illustrating load balancing methods;

FIG. 7 is a control flow chart illustrating a failover method;

FIG. 8 illustrates hardware and software components of a client/server in the clustered enterprise Java™ architecture shown in FIGS. 3–5.

The invention will be better understood with reference to the drawings and detailed description below. In the drawings, like reference numerals indicate like components.

DETAILED DESCRIPTION

I. Clustered Enterprise Java™ Distributed Processing System

A. Clustered Enterprise Java™ Software Architecture

FIG. 3*a* illustrates a simplified block diagram **380** of the software layers in a processing device of a clustered enterprise Java™ system, according to an embodiment of the present invention. A detailed description of a clustered enterprise Java™ distributed processing system is described below. The first layer of software includes a communication medium software driver **351** for transferring and receiving information on a communication medium, such as an ethernet local area network. An operating system **310** including a transmission control protocol ("TCP") software component **353** and internet protocol ("IP") software component **352** are upper software layers for retrieving and sending packages or blocks of information in a particular format. An "upper" software layer is generally defined as a software component which utilizes or accesses one or more "lower" software layers or software components. A JVM **354** is then implemented. A kernel **355** having a remote Java™ virtual machine **356** is then layered on JVM **354**. Kernel **355**. described in detail below, is used to transfer messages between processing devices in a clustered enterprise Java™ distributed processing system. Remote method invocation **357** and enterprise Java™ bean **358** are upper software layers of kernel **355**. EJB **358** is a container for a variety of Java™ applications.

FIG. 3*b* illustrates a detailed view of kernel **355** illustrated in FIG. 3*a*. Kernel **355** includes a socket manager component **363**, thread manager **364** component, and RJVM **356**. RJVM **356** is a data structure including message routing software component **360**, message compression software component **361** including abbreviation table **161***c*, and peer-gone detection software component **362**. RJVM **356** and thread manager component **364** interact with socket manager component **363** to transfer information between processing devices.

B. Distributed Processing System

FIG. 3 illustrates a simplified block diagram of a clustered enterprise Java™ distributed processing system **300**. Processing devices are coupled to communication medium **301**. Communication medium **301** may be a wired and/or wireless communication medium or combination thereof. In an embodiment, communication medium **301** is a local-area-network (LAN). In an alternate embodiment, communication medium **301** is a world-area-network (WAN) such as the Internet or World Wide Web. In still another embodiment, communication medium **301** is both a LAN and a WAN.

A variety of different types of processing devices may be coupled to communication medium **301**. In an embodiment, a processing device may be a general purpose computer **100** as illustrated in FIG. **8** and described below. One of ordinary skill in the art would understand that FIG. **8** and the below description describes one particular type of processing device where multiple other types of processing devices with a different software and hardware configurations could be utilized in accordance with an embodiment of the present invention. In an alternate embodiment, a processing device may be a printer, handheld computer, laptop computer, scanner, cellular telephone, pager, or equivalent thereof.

FIG. 3*c* illustrates an embodiment of the present invention in which servers **302** and **303** are coupled to communication medium **301**. Server **303** is also coupled to communication medium **305** which may have similar embodiments as described above in regard to communication medium **301**. Client **304** is also coupled to communication medium **305**. In an alternate embodiment, client **304** may be coupled to communication medium **301** as illustrated by the dashed line and box in FIG. 3*c*. It should be understood that in alternate embodiments, server **302** is (1) both a client and a server, or (2) a client. Similarly, FIG. 3 illustrates an embodiment in which three processing devices are shown wherein other embodiments of the present invention include multiple other processing devices or communication mediums as illustrated by the ellipses.

Server **302** transfers information over communication medium **301** to server **303** by using network software **302***a* and network software **303***a*, respectively. In an embodiment, network software **302***a*, **303***a*, and **304***a* include communication medium software driver **351**, Transmission Control Protocol software **353**, and Internet Protocol software **352** ("TCP/IP"). Client **304** also includes network software **304***a* for transferring information to server **303** over communication medium **305**. Network software **303***a* in server **303** is also used to transfer information to client **304** by way of communication medium **305**.

According to an embodiment of the present invention, each processing device in clustered enterprise Java™ architecture **300** includes a message-passing kernel **355** that supports both multi-tier and peer-to-peer functionality. A kernel is a software program used to provide fundamental services to other software programs on a processing device.

In particular, server **302**, server **303**, and client **304** have kernels **302***b*, **303***b*, and **304***b*, respectively. In particular, in order for two JVMs to interact, whether they are clients or servers, each JVM constructs an RJVM representing the other. Messages are sent from the upper layer on one side, through a corresponding RJVM, across the communication medium, through the peer RJVM, and delivered to the upper layer on the other side. In various embodiments, messages can be transferred using a variety of different protocols, including, but not limited to, Transmission Control Protocol/Internet Protocol ("TCP/IP"), Secure Sockets Layer ("SSL"), Hypertext Transport Protocol ("HTTP") tunneling, and Internet InterORB Protocol ("IIOP") tunneling, and combinations thereof. The RJVMs and socket managers create and maintain the sockets underlying these protocols and share them between all objects in the upper layers. A socket is a logical location representing a terminal between processing devices in a distributed processing system. The kernel maintains a pool of execute threads and thread manager software component **364** multiplexes the threads between socket reading and request execution. A thread is a sequence of executing program code segments or functions.

For example, server **302** includes JVM1 and Java™ application **302***c*. Server **302** also includes a RJVM2 representing the JVM2 of server **303**. If a message is to be sent from server **302** to server **303**, the message is sent through RJVM2 in server **302** to RJVM1 in server **303**.

C. Message Forwarding

Clustered enterprise Java™ network **300** is able to forward a message through an intermediate server. This functionality is important if a client requests a service from a back-end server through a front-end gateway. For example, a message from server **302** (client **302**) and, in particular, JVM1 may be forwarded to client **304** (back-end server **304**) or JVM3 through server **303** (front-end gateway) or JVM2. This functionality is important in controlling session concentration or how many connections are established between a server and various clients.

Further, message forwarding may be used in circumstances where a socket cannot be created between two JVMs. For example, a sender of a message is running under the applet security model which does not allow for a socket to be created to the original server. A detailed description of the applet security model is provided at http//:www.javasoft.com, which is incorporated herein by reference. Another example includes when the receiver of the message is behind a firewall. Also, as described below, message forwarding is applicable if the sender is a client and the receiver is a client and thus does not accept incoming sockets.

For example, if a message is sent from server **302** to client **304**, the message would have to be routed through server **303**. In particular, a message handoff, as illustrated by **302***f*, between RJVM3 (representing client **304**) would be made to RJVM2 (representing server **303**) in server **302**. The message would be transferred using sockets **302***e* between RJVM2 in server **302** and RJVM1 in server **303**. The message would then be handed off, as illustrated by dashed line **303***f*, from RJVM1 to RJVM3 in server **303**. The message would then be passed between sockets of RJVM3 in server **303** and RJVM2 in client **304**. The message then would be passed, as illustrated by the dashed line **304***f*, from RJVM2 in client **304** to RJVM1 in client **304**.

D. Rerouting

An RJVM in client/server is able to switch communication paths or communication mediums to other RJVMs at any time. For example, if client **304** creates a direct socket to server **302**, server **302** is able to start using the socket instead of message forwarding through server **303**. This embodiment is illustrated by a dashed line and box representing client **304**. In an embodiment, the use of transferring messages by RJVMs ensures reliable, in-order message delivery after the occurrence of a network reconfiguration. For example, if client **304** was reconfigured to communication medium **301** instead of communication medium **305** as illustrated in FIG. **3**. In an alternate embodiment, messages may not be delivered in order.

An RJVM performs several end-to-end operations that are carried through routing. First, an RJVM is responsible for detecting when a respective client/server has unexpectedly died. In an embodiment, peer-gone selection software component **362**, as illustrated in FIG. **3***b*, is responsible for this function. In an embodiment, an RJVM sends a heartbeat message to other clients/servers when no other message has been sent in a predetermined time period. If the client/server does not receive a heartbeat message in the predetermined count time, a failed client/server which should have sent the heartbeat, is detected. In an embodiment, a failed client/server is detected by connection timeouts or if no messages have been sent by the failed client/server in a predetermined amount of time. In still another embodiment, a failed socket indicates a failed server/client.

Second, during message serialization, RJVMs, in particular, message compression software **360**, abbreviate commonly transmitted data values to reduce message size. To accomplish this, each JVM/RJVM pair maintains matching abbreviation tables. For example, JVM1 includes an abbreviation table and RJVM1 includes a matching abbreviation table. During message forwarding between an intermediate server, the body of a message is not deserialized on the intermediate server in route.

E. Multi-tier/Peer-to-Peer Functionality

Clustered enterprise Java™ architecture **300** allows for multitier and peer-to-peer programming.

Clustered enterprise Java™ architecture **300** supports an explicit syntax for client/server programming consistent with a multitier distributed processing architecture. As an example, the following client-side code fragment writes an informational message to a server's log file:

```
T3Client clnt=new T3Client("t3://acme:7001");
LogServices log=clnt.getT3Services( ).log( );
log.info("Hello from a client");
```

The first line establishes a session with the acme server using the t3 protocol. If RJVMs do not already exist, each JVM constructs an RJVM for the other and an underlying TCP socket is established. The client-side representation of this session—the T3Client object—and the server-side representation communicate through these RJVMs. The server-side supports a variety of services, including database access, remote file access, workspaces, events, and logging. The second line obtains a LogServices object and the third line writes the message.

Clustered enterprise Java™ computer architecture **300** also supports a server-neutral syntax consistent with a peer-to-peer distributed processing architecture. As an example, the following code fragment obtains a stub for an RMI object from the JNDI-compliant naming service on a server and invokes one of its methods.

```
Hashtable env=new Hashtable( );
env.put(Context.PROVIDER_URL, "t3://acme:7001");
env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WebLogicinitialContextFactory");
Context ctx=new InitialContext(env);
Example e=(Example) ctx.lookup("acme.eng.example");
result=e.example(37);
```

In an embodiment, JNDI naming contexts are packaged as RMI objects to implement remote access. Thus, the above code illustrates a kind of RMI bootstrapping. The first four lines obtain an RMI stub for the initial context on the acme server. If RJVMs do not already exist, each side constructs an RJVM for the other and an underlying TCP socket for the t3 protocol is established. The caller-side object—the RMI stub—and the callee-side object—an RMI impl-communicate through the RJVMs. The fifth line looks up another RMI object, an Example, at the name acme.eng.example and the sixth line invokes one of the Example methods. In an embodiment, the Example impl is not on the same processing device as the naming service. In another embodiment, the Example impl is on a client. Invocation of the Example object leads to the creation of the appropriate RJVMs if they do not already exist.

II. Replica-Aware or Smart Stubs/EJB Objects

In FIG. **3***c*, a processing device is able to provide a service to other processing devices in architecture **300** by replicating RMI and/or EJB objects. Thus, architecture **300** is easily scalable and fault tolerant. An additional service may easily be added to architecture **300** by adding replicated RMI and/or EJB objects to an existing processing device or newly added processing device. Moreover, because the RMI and/or

EJB objects can be replicated throughout architecture **300**, a single processing device, multiple processing devices, and/or a communication medium may fail and still not render architecture **300** inoperable or significantly degraded.

FIG. **5***a* illustrates a replica-aware ("RA") or Smart stub **580** in architecture **500**. Architecture **500** includes client **504** coupled to communication medium **501**. Servers **502** and **503** are coupled to communication medium **501**, respectively. Persistent storage device **509** is coupled to server **502** and **503** by communication medium **560** and **561**, respectively. In various embodiments, communication medium **501**, **560**, and **561** may be wired and/or wireless communication mediums as described above. Similarly, in an embodiment, client **504**, server **502**, and server **503** may be both clients and servers as described above. One of ordinary skill in the art would understand that in alternate embodiments, multiple other servers and clients may be included in architecture **500** as illustrated by ellipses. Also, as stated above, in alternate embodiments, the hardware and software configuration of client **504**, server **502** and server **503** is described below and illustrated in FIG. **8**.

RA RMI stub **580** is a Smart stub which is able to find out about all of the service providers and switch between them based on a load balancing method **507** and/or failover method **508**. In an embodiment, an RA stub **580** includes a replica handler **506** that selects an appropriate load balancing method **507** and/or failover method **507**. In an alternate embodiment, a single load balancing method and/or single failover method is implemented. In alternate embodiments, replica handler **506** may include multiple load balancing methods and/or multiple failover methods and combinations thereof. In an embodiment, a replica handler **506** implements the following interface:

```
public interface ReplicaHandler {
  Object loadBalance(Object currentProvider) throws
RefreshAbortedException;
  Object failOver(Object failedProvider,
  RemoteException e) throws
RemoteException;
}
```

Immediately before invoking a method, RA stub **580** calls load balance method **507**, which takes the current server and returns a replacement. For example, client **504** may be using server **502** for retrieving data for database **509***a* or personal storage device **509**. Load balance method **507** may switch to server **503** because server **502** is overloaded with service requests. Handler **506** may choose a server replacement entirely on the caller, perhaps using information about server **502** load, or handler **506** may request server **502** for retrieving a particular type of data. For example, handler **506** may select a particular server for calculating an equation because the server has enhanced calculation capability. In an embodiment, replica handler **506** need not actually switch providers on every invocation because replica handler **506** is trying to minimize the number of connections that are created.

FIG. **6***a* is a control flow diagram illustrating the load balancing software **507** illustrated in FIGS. **5***a–b*. It should be understood that FIG. **6***a* is a control flow diagram illustrating the logical sequence of functions or steps which are completed by software in load balancing method **507**. In alternate embodiments, additional functions or steps are completed. Further, in an alternate embodiment, hardware may perform a particular function or all the functions.

Load balancing software **507** begins as indicated by circle **600**. A determination is then made in logic block **601** as to whether the calling thread established "an affinity" for a particular server. A client has an affinity for the server that coordinates its current transaction and a server has an affinity for itself. If an affinity is established, control is passed to logic block **602**, otherwise control is passed to logic block **604**. A determination is made in logic block **602** whether the affinity server provides the service requested. If so, control is passed to logic block **603**. Otherwise, control is passed to logic block **604**. The provider of the service on the affinity server is returned to the client in logic block **603**. In logic block **604**, a naming service is contacted and an updated list of the current service providers is obtained. A getNextProvider method is called to obtain a service provider in logic block **605**. Various embodiments of the getNextProvider method are illustrated in FIGS. **6***b–g* and described in detail below. The service is obtained in logic block **606**. Failover method **508** is then called if service is not provided in logic block **606** and load balancing method **507** exits as illustrated by logic block **608**. An embodiment of failover method **508** is illustrated in FIG. **7** and described in detail below.

FIGS. **6***b–g* illustrate various embodiments of a getNextProvider method used in logic block **605** of FIG. **6***a*. As illustrated in FIG. **6***b*, the getNextProvider method selects a service provider in a round robin manner. A getNextProvider method **620** is entered as illustrated by circle **621**. A list of current service providers is obtained in logic block **622**. A pointer is incremented in logic block **623**. The next service provider is selected based upon the pointer in logic block **624** and the new service provider is returned in logic block **625** and getNextProvider method **620** exits as illustrated by circle **626**.

FIG. **6***c* illustrates an alternate embodiment of a getNextProvider method which obtains a service provider by selecting a service provider randomly. A getNextProvider method **630** is entered as illustrated by circle **631**. A list of current service providers is obtained as illustrated by logic block **632**. The next service provider is selected randomly as illustrated by logic block **633** and a new service provider is returned in logic block **634**. The getNextProvider method **630** then exits, as illustrated by circle **635**.

Still another embodiment of a getNextProvider method is illustrated in FIG. **6***d* which obtains a service provider based upon the load of the service providers. A getNextProvider method **640** is entered as illustrated by circle **641**. A list of current service providers is obtained in logic block **642**. The load of each service provider is obtained in logic block **643**. The service provider with the least load is then selected in logic block **644**. The new service provider is then returned in logic block **645** and getNextProvider method **640** exits as illustrated by circle **646**.

An alternate embodiment of a getNextProvider method is illustrated in FIG. **6***e* which obtains a service provider based upon the type of data obtained from the service provider. A getNextProvider method **650** is entered as illustrated by circle **651**. A list of current service providers is obtained in logic block **652**. The type of data requested from the service providers is determined in logic block **653**. The service provider is then selected based on the data type in logic block **654**. The service provider is returned in logic block **655** and getNextProvider method **650** exits as illustrated by circle **656**.

Still another embodiment of a getNextProvider method is illustrated in FIG. **6***f* which selects a service provider based upon the physical location of the service providers. A getNextProvider method **660** is entered as illustrated by circle **661**. A list of service providers is obtained as illustrated by logic block **662**. The physical distance to each service provider is determined in logic block **663** and the

service provider which has the closest physical distance to the requesting client is selected in logic block **664**. The new service provider is then returned in logic block **665** and the getNextProvider method **660** exits as illustrated by circle **666**.

Still a further embodiment of the getNextProvider method is illustrated in FIG. **6***g* and selects a service provider based on the amount of time taken for the service provider to respond to previous requests. Control of getNextProvider method **670** is entered as illustrated by circle **671**. A list of current service providers is obtained in logic block **672**. The time period for each service provider to respond to a particular message is determined in logic block **673**. The service provider which responds in the shortest time period is selected in logic block **674**. The new service provider is then returned in logic block **675** and control from getNextProvider method **670** exits as illustrated by circle **676**.

If invocation of a service method fails in such a way that a retry is warranted, RA **580** stub calls failover method **508**, which takes the failed server and an exception indicating what the failure was and returns a new server for the retry. If a new server is unavailable, RA stub **580** throws an exception.

FIG. **7** is a control flow chart illustrating failover software **508** shown in FIGS. **5***a–b*. Failover method **508** is entered as illustrated by circle **700**. A failed provider from the list of current providers of services is removed in logic block **701**. A getNextProvider method is then called in order to obtain a service provider. The new service provider is then returned in logic block **703** and failover method **508** exits as illustrated by circle **704**.

While FIGS. **6–7** illustrate embodiments of a replica handler **506**, alternate embodiments include the following functions or combinations thereof implemented in a round robin manner.

First, a list of servers or service providers of a service is maintained. Whenever the list needs to be used and the list has not been recently updated, handler **506** contacts a naming service as described below and obtains an up-to-date list of providers.

Second, if handler **506** is about to select a provider from the list and there is an existing RJVM-level connection to the hosting server over which no messages have been received during the last heartbeat period, handler **506** skips that provider. In an embodiment, a server may later recover since death of peer is determined after several such heartbeat periods. Thus, load balancing on the basis of server load is obtained.

Third, when a provider fails, handler **506** removes the provider from the list. This avoids delays caused by repeated attempts to use non-working service providers.

Fourth, if a service is being invoked from a server that hosts a provider of the service, then that provider is used. This facilitates co-location of providers for chained invokes of services.

Fifth, if a service is being invoked within the scope of a transaction and the server acting as transaction coordinator hosts a provider of the service, then that provider is used. This facilitates co-location of providers within a transaction.

The failures that can occur during a method invocation may be classified as being either (1) application-related, or (2) infrastructure-related. RA stub **580** will not retry an operation in the event of an application-related failure, since there can be no expectation that matters will improve. In the event of an infrastructure-related failure, RA stub **580** may or may not be able to safely retry the operation. Some initial non-idempotent operation, such as incrementing the value of a field in a database, might have completed. In an embodiment, RA stub **580** will retry after an infrastructure failure only if either (1) the user has declared that the service methods are idempotent, or (2) the system can determine that processing of the request never started. As an example of the latter, RA stub **580** will retry if, as part of load balancing method, stub **580** switches to a service provider whose host has failed. As another example, a RA stub **580** will retry if it gets a negative acknowledgment to a transactional operation.

A RMI compiler recognizes a special flag that instructs the compiler to generate an RA stub for an object. An additional flag can be used to specify that the service methods are idempotent. In an embodiment, RA stub **580** will use the replica handler described above and illustrated in FIG. **5***a*. An additional flag may be used to specify a different handler. In addition, at the point a service is deployed, i.e., bound into a clustered naming service as described below, the handler may be overridden.

FIG. **5***b* illustrates another embodiment of the present invention in which an EJB object **551** is used instead of a stub, as shown in FIG. **5***a*.

III. Replicated JNDI-compliant Naming Service

As illustrated in FIG. **4**, access to service providers in architecture **400** is obtained through a JNDI-compliant naming service, which is replicated across architecture **400** so there is no single point of failure. Accordingly, if a processing device which offers a JNDI-compliant naming service fails, another processing device having a replicated naming service is available. To offer an instance of a service, a server advertises a provider of the service at a particular node in a replicated naming tree. In an embodiment, each server adds a RA stub for the provider to a compatible service pool stored at the node in the server's copy of the naming tree. If the type of a new offer is incompatible with the type of offers in an existing pool, the new offer is made pending and a callback is made through a ConflictHandler interface. After either type of offer is retracted, the other will ultimately be installed everywhere. When a client looks up the service, the client obtains a RA stub that contacts the service pool to refresh the client's list of service providers.

FIG. **4** illustrates a replicated naming service in architecture **400**. In an embodiment, servers **302** and **303** offer an example service provider P1 and P2, respectively, and has a replica of the naming service tree **402** and **403**, respectively. The node acme.eng.example in naming service tree **402** and **403** has a service pool **402***a* and **403***a*, respectively, containing a reference to Example service provider P1 and P2. Client **304** obtains a RA stub **304***e* by doing a naming service lookup at the acme.eng.example node. Stub **304***e* contacts an instance of a service pool to obtain a current list of references to available service providers. Stub **304***e* may switch between the instances of a service pool as needed for load-balancing and failover.

Stubs for the initial context of the naming service are replica-aware or Smart stubs which initially load balance among naming service providers and switch in the event of a failure. Each instance of the naming service tree contains a complete list of the current naming service providers. The stub obtains a fresh list from the instance it is currently using. To bootstrap this process, the system uses Domain Naming Service ("DNS") to find a (potentially incomplete) initial list of instances and obtains the complete list from one of them. As an example, a stub for the initial context of the naming service can be obtained as follows:

```
Hashtable env=new Hashtable( );
env.put(Context.PROVIDER_URL, "t3://
  acmeCluster:7001");
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WebLogicInitialContextFactor");
Context ctx=new InitialContext(env);
```

Some subset of the servers in an architecture have been bound into DNS under the name acmeCluster. Moreover, an application is still able to specify the address of an individual server, but the application will then have a single point of failure when the application first attempts to obtain a stub.

A reliable multicast protocol is desirable. In an embodiment, provider stubs are distributed and replicated naming trees are created by an IP multicast or point-to-point protocol. In an IP multicast embodiment, there are three kinds of messages: Heartbeats, Announcements, and StateDumps. Heartbeats are used to carry information between servers and, by their absence, to identify failed servers. An Announcement contains a set of offers and retractions of services. The Announcements from each server are sequentially numbered. Each receiver processes an Announcement in order to identify lost Announcements. Each server includes in its Heartbeats the sequence number of the last Announcement it has sent. Negative Acknowledgments ("NAKs") for a lost Announcement are included in subsequent outgoing Heartbeats. To process NAKs, each server keeps a list of the last several Announcements that the server has sent. If a NAK arrives for an Announcement that has been deleted, the server sends a StateDump, which contains a complete list of the server's services and the sequence number of its next Announcement. When a new server joins an existing architecture, the new server NAKs for the first message from each other server, which results in StateDumps being sent. If a server does not receive a Heartbeat from another server after a predetermined period of time, the server retracts all services offered by the server not generating a Heartbeat.

IV. Programming Models

Applications used in the architecture illustrated in FIGS. **3–5** use one of three basic programming models: (1) stateless or direct, (2) stateless factory or indirect, or (3) stateful or targeted, depending on the way the application state is to be treated. In the stateless model, a Smart stub returned by a naming-service lookup directly references service providers.

```
Example e=(Example) ctx.lookup("acme.eng.example");
result1=e.example(37);
result2=e.example(38);
```

In this example, the two calls to example may be handled by different service providers since the Smart stub is able to switch between them in the interests of load balancing. Thus, the Example service object cannot internally store information on behalf of the application. Typically the stateless model is used only if the provider is stateless. As an example, a pure stateless provider might compute some mathematical function of its arguments and return the result. Stateless providers may store information on their own behalf, such as for accounting purposes. More importantly, stateless providers may access an underlying persistent storage device and load application state into memory on an as-needed basis. For example, in order for example to return the running sum of all values passed to it as arguments, example might read the previous sum from a database, add in its current argument, write the new value out, and then return it. This stateless service model promotes scalability.

In the stateless factory programming model, the Smart stub returned by the lookup is a factory that creates the desired service providers, which are not themselves Smart stubs.

```
ExampleFactory gf=(ExampleFactory)
ctx.lookup("acme.eng.example");
Example e=gf.create( );
result1=e.example(37);
result2=e.example(38);
```

In this example, the two calls to example are guaranteed to be handled by the same service provider. The service provider may therefore safely store information on behalf of the application. The stateless factory model should be used when the caller needs to engage in a "conversation" with the provider. For example, the caller and the provider might engage in a back-and-forth negotiation. Replica-aware stubs are generally the same in the stateless and stateless factory models, the only difference is whether the stubs refer to service providers or service provider factories.

A provider factory stub may failover at will in its effort to create a provider, since this operation is idempotent. To further increase the availability of an indirect service, application code must contain an explicit retry loop around the service creation and invocation.

```
while (true) {
  try {
    Example e=gf.create( );
    result1:=e.example(37);
    result2=e.example(38);
    break;
  } catch (Exception e) {
    if (!retryWarranted(e))
      throw e;
  }
}
```

This would, for example, handle the failure of a provider e that was successfully created by the factory. In this case, application code should determine whether non-idempotent operations completed. To further increase availability, application code might attempt to undo such operations and retry.

In the stateful programming model, a service provider is a long-lived, stateful object identified by some unique system-wide key. Examples of "entities" that might be accessed using this model include remote file systems and rows in a database table. A targeted provider may be accessed many times by many clients, unlike the other two models where each provider is used once by one client. Stubs for targeted providers can be obtained either by direct lookup, where the key is simply the naming-service name, or through a factory, where the key includes arguments to the create operation. In either case, the stub will not do load balancing or failover. Retries, if any, must explicitly obtain the stub again.

There are three kinds of beans in EJB, each of which maps to one of the three programming models. Stateless session beans are created on behalf of a particular caller, but maintain no internal state between calls. Stateless session beans map to the stateless model. Stateful session beans are created on behalf of a particular caller and maintain internal state between calls. Stateful session beans map to the stateless factory model. Entity beans are singular, stateful objects identified by a system-wide key. Entity beans map to the stateful model. All three types of beans are created by a factory called an EJB home. In an embodiment, both EJB homes and the beans they create are referenced using RMI. In an architecture as illustrated in FIGS. **3–5**, stubs for an EJB home are Smart stubs. Stubs for stateless session beans are Smart stubs, while stubs for stateful session beans and entity beans are not. The replica handler to use for an EJB-based service can be specified in its deployment descriptor.

To create an indirect RMI-based service, which is required if the object is to maintain state on behalf of the caller, the application code must explicitly construct the factory. A targeted RMI-based service can be created by running the RMI compiler without any special flags and then binding the resulting service into the replicated naming tree. A stub for the object will be bound directly into each instance of the naming tree and no service pool will be created. This provides a targeted service where the key is the naming-service name. In an embodiment, this is used to create remote file systems.

V. Hardware and Software Components

FIG. **8** shows hardware and software components of an exemplary server and/or client as illustrated in FIGS. **3–5**. The system of FIG. **8** includes a general-purpose computer **800** connected by one or more communication mediums, such as connection **829**, to a LAN **840** and also to a WAN, here illustrated as the Internet **880**. Through LAN **840**, computer **800** can communicate with other local computers, such as a file server **841**. In an embodiment, file server **801** is server **303** as illustrated in FIG. **3**. Through the Internet **880**, computer **800** can communicate with other computers, both local and remote, such as World Wide Web server **881**. In an embodiment, Web server **881** is server **303** as illustrated in FIG. **3**. As will be appreciated, the connection from computer **800** to Internet **880** can be made in various ways, e.g., directly via connection **829**, or through local-area network **840**, or by modem (not shown).

Computer **800** is a personal or office computer that can be, for example, a workstation, personal computer, or other single-user or multi-user computer system; an exemplary embodiment uses a Sun SPARC-20 workstation (Sun Microsystems, Inc., Mountain View, Calif.). For purposes of exposition, computer **800** can be conveniently divided into hardware components **801** and software components **802**; however, persons of ordinary skill in the art will appreciate that this division is conceptual and somewhat arbitrary, and that the line between hardware and software is not a hard and fast one. Further, it will be appreciated that the line between a host computer and its attached peripherals is not a hard and fast one, and that in particular, components that are considered peripherals of some computers are considered integral parts of other computers. Thus, for example, user I/O **820** can include a keyboard, a mouse, and a display monitor, each of which can be considered either a peripheral device or part of the computer itself, and can further include a local printer, which is typically considered to be a peripheral. As another example, persistent storage **808** can include a CD-ROM (compact disc read-only memory) unit, which can be either peripheral or built into the computer.

Hardware components **801** include a processor (CPU) **805**, memory **806**, persistent storage **808**, user I/O **820**, and network interface **825** which are coupled to bus **810**. These components are well understood by those of skill in the art and, accordingly, need be explained only briefly here.

Processor **805** can be, for example, a microprocessor or a collection of microprocessors configured for multiprocessing.

Memory **806** can include read-only memory (ROM), randomaccess memory (RAM), virtual memory, or other memory technologies, singly or in combination. Persistent storage **808** can include, for example, a magnetic hard disk, a floppy disk, or other persistent read-write data storage technologies, singly or in combination. It can further include mass or archival storage, such as can be provided by CD-ROM or other large-capacity storage technology. (Note that file server **841** provides additional storage capability that processor **805** can use.)

User I/O (input/output) hardware **820** typically includes a visual display monitor such as a CRT or flat-panel display, an alphanumeric keyboard, and a mouse or other pointing device, and optionally can further include a printer, an optical scanner, or other devices for user input and output.

Network I/O hardware **825** provides an interface between computer **800** and the outside world. More specifically, network I/O **825** lets processor **805** communicate via connection **829** with other processors and devices through LAN **840** and through the Internet **880**.

Software components **802** include an operating system **850** and a set of tasks under control of operating system **310**, such as a Java™ application program **860** and, importantly, JVM software **354** and kernel **355**. Operating system **310** also allows processor **805** to control various devices such as persistent storage **808**, user I/O **820**, and network interface **825**. Processor **805** executes the software of operating system **310**, application **860**, JVM **354** and kernel **355** in conjunction with memory **806** and other components of computer system **800**. In an embodiment, software **802** includes network software **302***a*, JVM1, RJVM2 and RJVM3, as illustrated in server **302** of FIG. **3***c*. In an embodiment, Java™ application program **860** is Java™ application **302***c* as illustrated in FIG. **3***c*.

Persons of ordinary skill in the art will appreciate that the system of FIG. **8** is intended to be illustrative, not restrictive, and that a wide variety of computational, communications, and information devices can be used in place of or in addition to what is shown in FIG. **8**. For example, connections through the Internet **880** generally involve packet switching by intermediate router computers (not shown), and computer **800** is likely to access any number of Web servers, including but by no means limited to computer **800** and Web server **881**, during a typical Web client session.

The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in the art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical applications, thereby enabling others skilled in the art to understand the invention for various embodiments and with the various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

1. An article of manufacture including an information storage medium wherein is stored information, comprising:

a first set of digital information, including a Java™ virtual machine with a stub having a load balancing software component for selecting a service provider from a plurality of service providers and a failover software component for removing a failed service provider from a list identifying the plurality of service providers, wherein the Java™ virtual machine with the stub is located on a client processing device,

wherein the load balancing software selects a particular service provider, from the list of plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in a transaction between either of the service provider or server and the client processing device.

2. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of plurality of service providers in a round robin manner.

3. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component randomly selects a service provider from the list of service providers.

4. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the load of each service provider.

5. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the data type requested.

6. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the closest physical service provider.

7. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon a time period in which each service provider responds.

8. An article of manufacture including an information storage medium wherein is stored information, comprising:

a first set of digital information, including a Java™ virtual machine with a Java™ bean object for selecting a service provider from a plurality of service providers;

wherein the Java™ bean object has a load balancing software component that selects a particular service provider, from the plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in a transaction between either of the service provider or server and the client processing device.

9. The article of manufacture of claim 8, wherein the Java™ bean object has a failover software component for removing a failed service provider from a list of service providers.

10. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers in a round robin manner.

11. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component randomly selects a service provider.

12. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the load of each service provider.

13. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the data type requested.

14. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the closest physical service provider.

15. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon a time period in which each service provider responds.

16. The article of manufacture of claim 8, further comprising:

a second set of digital information, including a stateless session bean.

17. The article of manufacture of claim 8, further comprising:

a second set of digital information, including a stateful session bean.

18. The article of manufacture of claim 8, further comprising:

a second set of digital information, including an entity session bean.

19. An apparatus, comprising:

a processor;

an instruction store, coupled to the processor, comprising an article of manufacture as recited in claim 1; and

a data store, coupled to the processor, wherein an application program can be stored.

20. An apparatus, comprising:

a processor;

an instruction store, coupled to the processor, comprising an article of manufacture as recited in claim 8; and

a data store, coupled to the processor, wherein an application program can be stored.

21. A processing device implemented method, comprising the steps of:

obtaining, by a stub, a list of service providers; and

selecting a service provider for use in a transaction, by the stub, from the list of service providers

wherein the selecting step includes selecting, by the stub, a particular service provider, from the list of plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server

associated with the service provider, is currently participating in the transaction.

22. The method of claim 21, wherein the list of service providers is obtained from a naming service.

23. The method of claim 21, wherein the list of service providers is obtained from a naming service.

24. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the step of:

selecting a service provider, by the stub, from the list of service providers in a round robin manner.

25. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the step of:

selecting a service provider, by the stub, randomly from the list of service providers.

26. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

obtaining the load of each service provider, by the stub, in the list of service providers; and,

selecting a service provider, by the stub, based upon the load of each service provider.

27. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining the type of data requested; and,

selecting a service provider, by the stub, from the list of service providers based upon the data type.

28. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining the physical distance to each service provider, by the stub, in the list of service providers; and,

selecting a service provider, by the stub, from the list of service providers based upon on the closest physical distance to the service provider.

29. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining a time period for each service provider, by the stub, in the list of service providers to respond; and,

selecting a service provider from the list of service providers based upon the time period for each service provider to respond.

30. A processing device implemented method, comprising:

obtaining a calling thread;

determining, by a client, if the calling thread has an affinity for a server, wherein an affinity exists for a particular server when that particular server is currently participating in a transaction between the server and the client;

determining if the server provides a service;

obtaining a list of services, wherein the service is in the list of services providers; and,

attempting to obtain the service.

31. The method of claim 30, further comprising:

obtaining a failover method if the service is not available.

32. The method of claim 31, wherein the failover method obtains a next service provider in the list of service providers.

33. The method of claim 31, wherein the failover method obtains a randomly selected service provider in the list of service providers.

34. The method of claim 31, wherein the failover method obtains a service provider with the least load in the list of service providers.

35. The method of claim 31, wherein the failover method obtains a service provider based on a data type in the list of service providers.

36. The method of claim 31, wherein the failover method obtains a service provider based on the closest physical distance to the service provider.

37. The method of claim 31, wherein the failover method obtains a service provider based on a time for a response from a service provider in the plurality of service providers.

38. A method of providing failover in a distributed processing system, to select which service provider within a plurality of service providers should respond to a request from a client to access a service, the method comprising the steps of:

determining whether the client has an affinity for a particular service provider within the plurality of service providers;

if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of that service provider to the client;

if the client does not have an affinity for a particular service provider, or if the particular service provider is no longer available to provide the service requested, then the substeps of

selecting a new service provider from the plurality of service providers, and,

returning the name of the new service provider to the client; and,

allowing the client to access the service using the named service provider, if the named service provider exists and can provide the service.

39. The method of claim 38 wherein said step (B) of determining whether the client has an affinity for a particular service provider includes determining which service provider is coordinating the current transaction between the client and the server, and identifying that service provider as the particular service provider which the client has an affinity for.

40. The method of claim 38 wherein said step of selecting a new service provider includes selecting a new service provider from a list of service providers.

41. The method of claim 38 wherein said step of selecting a new service provider includes selecting a service provider from a list of service providers which is maintained by a naming service.

42. The method of claim 38 wherein said step of selecting a new service provider includes calling a get.next.provider function to obtain and select the next service provider on the list of service providers.

43. The method of claim 38 further comprising, following said step (D) of allowing the client to request service from the named service provider, the additional steps of:

(E) if the named service provider does not exist or cannot provide the service, then calling a failover method that includes the substeps of

identifying the named service provider as a failed service provider,

selecting a failover service provider from the plurality of service providers, and,

returning the name of the failover service provider to the client.

44. The method of claim 38 further comprising the step of:

removing the failed service provider from a list of available service providers within the distributed processing system.

45. The method of claim 38 wherein the service is a database or file system.

46. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

47. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

48. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the load of each service provider.

49. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

50. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

51. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

52. A method of using load balancing and/or failover in a distributed processing system to select which a service provider within a plurality of service providers can respond to a transaction request from a client to access a service, the method comprising the steps of:

(A) receiving a transaction request from a client to access a service;

(B) determining whether the client has an affinity for a particular service provider within said plurality of service providers;

(C1) if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of the particular service provider to the client for use by the client in accessing the service;

(C2) if the client does not have an affinity for a particular service provider, then the substeps of

selecting a new service provider from the plurality of service providers,

determining whether the new service provider can provide the service requested, and,

returning the name of the new service provider to the client for use by the client in accessing the service; and,

(D) allowing the client to request service from the named service provider, if the named service provider is available and can provide access to the service requested.

53. The method of claim 52 wherein said step (B) of determining whether the client has an affinity for a particular service provider includes determining which service provider is coordinating the current transaction between the client and the server, and identifying that service provider as the particular service provider which the client has an affinity for.

54. The method of claim 52 wherein said step of selecting a new service provider includes selecting a new service provider from a list of service providers.

55. The method of claim 52 wherein said step of selecting a new service provider includes selecting a service provider from a list of service providers which is maintained by a naming service.

56. The method of claim 52 wherein said step of selecting a new service provider includes calling a get.next.provider function to obtain and select the next service provider on the list of service providers.

57. The method of claim 52 further comprising, following said step (D) of allowing the client to request service from the named service provider, the additional steps of:

(E) if the named service provider does not exist or cannot provide the service, then calling a failover method that includes the substeps of

identifying the named service provider as a failed service provider,

selecting a failover service provider from the plurality of service providers, and,

returning the name of the failover service provider to the client.

58. The method of claim 52 further comprising the step of:

removing the failed service provider from a list of available service providers within the distributed processing system.

59. The method of claim 52 wherein the service is a database or file system.

60. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

61. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

62. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from

the list of service providers based upon the load of each service provider.

63. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

64. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

65. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

66. A system for load balancing and failover of requests by a client to access a service in a distributed processing system comprising:

a handler for receiving requests from a client to access a service;

a plurality of service providers for providing client access to the service;

software code that performs the method of

determining whether the client has an affinity for a particular service provider within the plurality of service providers;

if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of that service provider to the client;

if the client does not have an affinity for a service provider, then the substeps of selecting a new service provider from the plurality of service providers, and,

returning the name of the new service provider to the client; and,

allowing the client to access the service using the named service provider, if the named service provider exists and can provide the service.

67. The system of claim 66 further comprising:

a list of currently available service providers, and,

wherein a failed service provider is removed from the list of service providers.

68. The system of claim 66 wherein the service is a database or file system.

69. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

70. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

71. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the load of each service provider.

72. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

73. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

74. The system of claim 66, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,581,088 B1
DATED : June 17, 2003
INVENTOR(S) : Dean B. Jacobs and Eric M. Halpern

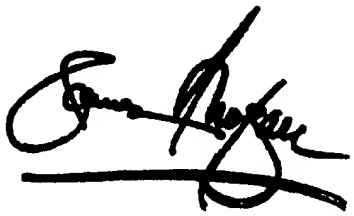
It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 21,
Lines 3 and 4, please replace Claim 22 with the following:

22. The method of claim 21, further comprising the step of: removing a failed service provider, by the stub, from the list of service providers.

Signed and Sealed this

Seventh Day of October, 2003



JAMES E. ROGAN
Director of the United States Patent and Trademark Office